PLATO
Final Report

Fall 2013

Daniel Perlmutter (dap2163) — Project Leader
Yasser Abhoudkhil (ya2282) Joaquín Ruales (jar2262)

Contents

1. Introduction

PLATO—Programing Language for Abstract Transformation Operators—is designed for succinctly expressing mathematical programs. It is inspired by Matlab, Python, Prolog, Java, and Ocaml.

PLATO is

- Compiled
- Strongly typed
- Imperative
- Not Object Oriented

High Level
PLATO is a high level language that allows complicated algorithms to be expressed succinctly.

Mathematical
PLATO includes an easy way to define mathematical concepts such as set and vectors or even groups and fields. PLATO's syntax eschews the use of the equals sign alone as an assignment operator and uses it as it was mathematically intended to represent equality. Since PLATO has no global variables each function starts with only the only values in scope being its parameters, more akin to the mathematical definition of a function than the computer science definition.

Readable
The use of ALL CAPS for built in functions and descriptive operator names like “AND” instead of “&&” makes PLATO code easier to read.

Architecture Neutral, Portable
By using a Java backend we ensure that Plato programs can run wherever Java can.

2. Language Tutorial

As we all know, Cayley tables are great fun. However, what do you do when your finite groups are so large that your tables don't fit on your graph paper anymore?

PLATO (Programing Language for Abstract Transformation Operators) is a programming language inspired by Matlab, Python, Prolog, Java, and Ocaml, and designed for the direct manipulation of abstract mathematical expressions. In addition to being able to express basic number-theoretical algorithms, it can also also handle finite sets, groups, rings, and fields. In this tutorial, we'll get you started with PLATO.

2.1. A First Example

Lets start with the basics: here is how you calculate the greatest common divisor of two integers in PLATO using the Euclidian algorithm.

```
main() {
    INTEGER x := 2013;
    INTEGER y := 1612012015;
    PRINT gcd(x, y);
}
INTEGER gcd(INTEGER a, INTEGER b) {
    IF (b = 0) {
      RETURN a;
    } ELSE {
      RETURN gcd(b, a % b);
    }
}
```

Figure 2.1: Greatest common divisor in PLATO

The program starts executing at the `main()` function. We declare x to be an integer and we let it be equal to 2013. Likewise, we declare y to be an integer with value `1612012015`. We then print the value of the function `gcd` evaluated on these two previously defined integers.

The gcd function is defined below the end of main(). Let's look at its header:

```
INTEGER gcd(INTEGER a, INTEGER b)
```

This function is called `gcd`, its result will be of type `INTEGER`. The function expects two integers and upon receiving them it will be referring to them respectively as `a` and `b`. Let's now look at the body of this function:

```
IF (b = 0) {
  RETURN a;
} ELSE {
  RETURN gcd(b, a % b);
}
```

As we would expect, this function will return a result of `a` if `b` is zero, or a result of `gcd(b, a % b)` otherwise, where `%` denotes the modulo operator.

2.2. Compiling and running a PLATO program

In order to run our gcd program:

1. Save the program in Figure 2.1 into a file named GCD.plt
2. `cd` via terminal into the directory where GCD.plt is
3. To compile the program: `platoc GCD.plt` then `javac Main_GCD.java`
4. To run the program: `java Main_GCD`

2.3. Cases Syntax

Let's take a look at some of PLATO's features.
We'll first make use of PLATO's cases syntax to make our gcd function even more succinct.

```
main() {
  INTEGER x := 2013;
  INTEGER y := 1612012015;
  PRINT gcd(x, y);
}
INTEGER gcd(INTEGER a, INTEGER b) {
  RETURN {
    a                IF b=0;
    gcd(b, a % b)    OTHERWISE
  };
}
```

Figure 2.2: Greatest common divisor revisited

The cases syntax works just as you would expect it, it checks the IF expressions in order and returns the value to the left of the first one that is true. If none of the expressions are true, then the program returns the value to the left of OTHERWISE.

2.4. Sets and Vectors

What would number theory be without sets? PLATO allows us to create and manipulate finite sets with ease.

```
main() {
  SET<INTEGER> sunSet = {1, 2, 3};       /* set assignment */
  PRINT sunSet + {2, 3, 4};              /* union of sets */
  PRINT {5, 6, 7} ^ {6, 7, 8};           /* intersection of sets */
  PRINT {9, 10, 11} \ {10, 11, 12};      /* set difference */
  PRINT {{1,2},{2}} + {{2},{1,2,3}};     /* union of sets */
}
```

Figure 2.3: Operations on sets

Note that we have to specify what the set is of when we declare it, as we have done with SET<INTEGER> sunSet. Had we saved the set `{{1,2},{2}}` into a variable, it would have been of type `SET<SET<INTEGER>>`.

Additionally, we can use PLATO to create and manipulate finite vectors.

```
main() {
  VECTOR<INTEGER> myVector := [5, 4, 3];
  myVector := myVector @ [2, 1]    /* vector concatenation */
  myVector[2] := 9000;             /* vector assignment */
  VECTOR<BOOLEAN> myBoolVector := [TRUE, FALSE, FALSE, FALSE];
  myBoolVector[4] := TRUE;
  PRINT myVector;
  PRINT myBoolVector;
}
```

Figure 2.4: Basic operations on vectors

In Figure 2.4. we can see how to initialize, modify, and concatenate vectors. The next-to-last print statement prints out `[5, 9000, 3, 2, 1]`, whereas the last one prints out `[TRUE, FALSE, FALSE, TRUE]`.

Let's look at a more interesting example, shall we? The following program prints all prime numbers less than or equal to 100.

```
main() {
  PRINT primes(100);
}
VECTOR<INTEGER> sieve(VECTOR<INTEGER> v, INTEGER n) {
  if (v[1] ^ 2 <= n) {
    RETURN ([v[1]] @ sieve(v[NOT (v % v[1] = 0)], n));
  } else {
    RETURN v;
  }
}
VECTOR<INTEGER> primes(INTEGER x) {
  RETURN sieve([2 TO x], x);
}
```

Figure 2.4: Finding primes in PLATO

In this program, we start with a vector containing all numbers from 2 to x, and we use the Sieve of Eratosthenes to discard all composite numbers.

`[2 TO x]` generates the vector of integers from 2 to x, inclusive. `[v[1]]` accesses the first element in array `v` and stores it by itself in an array, and the `@` symbol concatenates two vectors.

The operation v%v[1]=0 applies the operator %v[1] to every element in v and immediately checks whether each item of the resulting vector is equal to zero. At this point we have a vector of booleans of the same size as v to which we apply a NOT operator, obtaining its boolean complement. We then access v in a special way, passing it not an integer index, but a boolean array of the same size as v. Doing this returns a subvector of v containing only the elements with indices equal to the indices where the passed boolean vector had a value of TRUE.

2.5. Groups, Rings, and Fields

As we have mentioned earlier, PLATO can handle groups, rings, and fields. Let's define some groups in PLATO

```
main() {
  NUMBER OVER GROUP z3 x := 2;
  PRINT x + x;                                  /* prints out 1 */
  PRINT 2 + 2;                                  /* prints out 4 */
  PRINT x - x - x;                              /* prints out 1 */
  PRINT 2 - 2 - 2;                              /* prints out -2 */
  NUMBER OVER GROUP z3plus1 y := 2;
  PRINT y + y;                                  /* prints out 3 */
}
GROUP z3 {
  SET<INTEGER> elements := {0, 1, 2};
  INTEGER add(INTEGER n1, INTEGER n2) {
    RETURN (n1 + n2) % 3;
  }
}
GROUP z3plus1 {
  SET<INTEGER> elements := {1, 2, 3};
  INTEGER add(INTEGER n1, INTEGER n2) {
    RETURN ((n1 - 1) + (n2 - 1)) % 3 + 1;
  }
}
```

Figure 2.3: Groups in PLATO

As you can see, to define a group, we need only define the elements in its set and its addition operation. PLATO computes the group's Cayley table to make sure this is in fact a group, throwing an error if it isn't. Furthermore, it calculates the inverse of each element so that inverses or subtraction don't need to be explicitly defined.

We close this tutorial with an example of how to create and manipulate rings and fields, which are defined very similarly to how we defined groups, with the added component of a multiply function. For fields, PLATO automatically creates a division function by precomputing the multiplicative inverse of each element.

```
main() {
  NUMBER OVER RING z3 x := 2;
  PRINT x * x;                          /* prints out 1 */
```

```
  PRINT 2 * 2;                          /* prints out 4 */
  NUMBER OVER FIELD z5 y := 2;
  NUMBER OVER FIELD z5 y2 := 3;
  PRINT y / y2;                         /* prints out 4 */
  PRINT 2 / 3;                          /* prints out 0 */
}
RING z3 {
  SET<INTEGER> elements := {0, 1, 2};
  INTEGER add(INTEGER n1, INTEGER n2) {
    RETURN (n1 + n2) % 3;
  }
  INTEGER multiply(INTEGER n1, INTEGER n2) {
    RETURN (n1 * n2) % 3;
  }
}
FIELD z5 {
  SET<INTEGER> elements := {0, 1, 2, 3, 4};
  INTEGER add(INTEGER n1, INTEGER n2) {
    RETURN (n1 + n2) % 5;
  }
  INTEGER multiply(INTEGER n1, INTEGER n2) {
    RETURN (n1 * n2) % 5;
  }
}
```

Figure 2.3: Rings and fields in PLATO

3. Language Manual

PLATO Language Reference Manual

Daniel Perlmutter (dap2163) -- Project Leader
Yasser Abhoudkhil (ya2282) Joaquín Ruales (jar2262)

1) Introduction

This document describes the PLATO programming language. There are seven sections to this manual:
1) This introduction
2) Tokens -- a description of each class of tokens
3) Types -- a brief description of each primitive and derived data type, including syntax
4) Expressions -- parts of statements which need to be evaluated during execution
5) Statements -- components of a PLATO program which are executed
6) Blocks and Scope -- information regarding scoping of identifiers
7) PLATO Programs -- The structure of PLATO programs
8) PLATO CFG - The context free grammar describing the syntax of PLATO

2) Tokens

There are 4 classes of tokens: identifiers, reserved words, constants and separators. Comments are ignored by the compiler. Whitespace is ignored by the compiler except in serving to separate tokens from each other.

2.1) Comments

Comments begin with an opening “/*” and end with a closing “*/”. They may span multiple lines and may be nested.

2.2) Constants

There are two types of constants, booleans and integers. Their syntax is described in section 3.

2.3) Identifiers

Identifiers are defined by a sequence of alphanumeric characters beginning with a letter.
For a mathematically precise and formal definition, refer to the PLATO CFG in section 8

2.4) Reserved Words

Keywords and operators are characters or sequences of characters that have special meaning in PLATO.
They cannot be used as identifiers.
For a mathematically precise and formal definition, refer to the PLATO CFG in section 8.

2.3.1) Keywords

Keywords can be typed in all upper or all lower case but not a mixture of cases.
The keywords in PLATO are: true, false, print, and, or, not, if, elseif, else, to, by

2.3.2) Types

Types can be typed in all upper or all lower case but not a mixture of cases.
The types in PLATO are: boolean, integer, number, vector<t> set<t> where t is recursively defined to be any type. The value t is called a type parameter.

2.3.3) Operators

The operators in PLATO are: “+”, “-”, “*”, “/”, “\”, “%”, “**”, “^”, “>”, “>=”, “<”, “<=”, “=”, “@”

2.5) Separators

Separators are individual characters with special meaning in PLATO
The separators are “{“, “}”, “[“, “]”, “(“, “)”, “,”, “;”

3) Types

PLATO is strongly-typed with variable types determined at declaration time.
PLATO has 2 primitive and 4 derived types

3.1) Primitive Types

The two primitive types in PLATO are booleans and numbers.

3.1.1) Booleans

A boolean can take the value TRUE or FALSE and is specified by one of those keywords.

3.1.2) Integers

An integer is a sequence of one or more digits. The first digit of an integer may not be ‘0’ unless it is exactly the integer ‘0.’
For a mathematically precise and formal definition, refer to the PLATO CFG in section 8.

3.2) Derived Types

The four derived types in PLATO are numbers, matrices, sets and tuples.

3.2.1) Numbers

Numbers use the same syntax as Integers but they must be over some group, ring or field. If the group, ring or field is not specified by the user it is assumed to be the mathematical group of Integers.

3.2.2) Vectors

A vector can be specified by a comma separated list expressions which evaluates to numbers or boolean, representing the elements of the vector. It can also be specified with the syntax *start* "TO" *end* optionally followed by "BY" *increment* where *start*, *end* and *increment* are expressions which evaluate to integers. The *start* "TO" *end* "BY" *increment* syntax is equivalent to a comma separated list of numbers containing exactly those values in the infinite sequence startVal, startVal + incrementVal, start + incrementVal + incrementVal that are less than or equal to endVal where startVal, endVal and incrementVal are the value that *start*, *end*, and *increment* evaluate to, respectively.

Vectors are indexed left to right with the first element being element 1.

3.2.3) Sets

A set consists of zero or more numbers all of which must be over the same group, ring or field. A set is defined by an opening "{" and closing "}" surrounding a comma separated list of expressions that evaluate numbers.

4) Expressions

Expressions in PLATO can be evaluated to produce a value which is a primitive or derived type.

4.1) Values

A value is any primitive or derived type. A value evaluates to itself.

4.2) Unary operators

PLATO contains the the unary operators "NOT", which takes a Boolean value and evaluates to its logical negation, and "-" which takes an integer and returns its arithmetic inverse.

4.3) Binary operators

PLATO has several classes of binary operators including, arithmetic combiners, boolean combiners, comparators and matrix combiners. All binary operators are written in infix notation such as expr1 binOp expr2. The result of evaluating a binary operator is the result of evaluating the expression on the left hand side of the operators followed by the expression on the right hand side of the operators and then combining the two values with the given operator.

4.3.1) Arithmetic combiners

The arithmetic combiners take two integers and return an integer. They are “+”, “-”, “*”, “^”, ‘/’ and ’%’. The first 3 represent the standard addition, subtraction, multiplication and exponentiation operators on integers and the last 2 represent the quotient and remainder of integer division respectively.

4.3.2) Boolean combiners

The two boolean combiners are “AND” and “OR”. They take two booleans and return a boolean. They represent logical conjunction and disjunction respectively.

4.3.3) Comparators

Comparators take two integers and return a boolean. They are “>”, “>=”, “<”, “<=”, “=” and “!=”. They represent greater than, greater than or equal to, less than , less than or equal to, equal to, not equal to respectively.

4.4) Cases Expressions

PLATO allows for expressions of the form {expr IF expr; expr OTHERWISE}, {expr IF expr; expr IF expr; expr OTHERWISE}, and so on, which allows for cases resembling a mathematical cases clause. The expressions immediately to the left of the IFs and the OTHERWISE (we'll call these returnable expressions) need to share the same type, and the type of the whole expression becomes that type. The expression immediately to the right of each IF (we'll call these boolean expressions) must be an expression that evaluates to a boolean value. The value of a cases expression will be the value of the returnable expression to the left of the first boolean expression that evaluates to true. If none of the boolean expressions evaluates to true, the returnable expression to the left of OTHERWISE is returned. Boolean expressions are evaluated in order and only up until the first one evaluating to TRUE, and only the returned returnable expression gets evaluated out of all returnable expressions.

4.5) Operating on groups and extended groups

Numbers share some operators with integers

4.5.1) Operating on groups

Any number over a group can the binary “+” and “-” operators. These three operators represent addition and subtraction over the mathematical group.

4.5.2) Operating on extended groups

In addition to the operators defined on groups, any number over a ring or field can use the binary “*” and operator which is defined a multiplication over the mathematical ring or field. Any number over a field can also use the “/” operator which represents division over the mathematical field.

4.6) Operating on vectors

A vector is indexed by the syntax *vectorIdentifer*[*indexDefinition*] where *vectordentifier* is the identifier of a vector and *indexDefinition* is either an expression that evaluates to an integer or a vector of booleans.

4.6.1) Numerical indexing

If the *indexDefinition* during vector indexing is an expression that evaluates to an integer, then the result of the indexing is the value at that index in the vector.

4.6.2) Logical indexing

If the *indexDefinition* is an expression which evaluates to a vector of booleans of the same dimensions as the indexed matrix then the result is a vector whose elements are those elements of the indexed vector for which the element at the corresponding coordinates in the indexing vector has the value TRUE, in the same order as in the indexed vector.

4.6.3) Unary operators on matrices

Any unary operator can be applied to a vector as long as the elements are of a type to which the unary operator can be applied. The result is a vector formed by applying the operator to the matrix, element-wise.

4.6.4) Vectorization

Binary operators may be applied to a matrix and primitive as long as the types of the elements are the same as the type of the primitive and the primitive is of a type to which the binary operator can be applied. The primitive will be treated as a matrix of dimensions equal to the given matrix whose elements are all identical and equal to the value of the primitive.

4.7) Operating on sets

The “+”, “^”, “\” and “*” operators may be used on sets and represent set union, set intersection, set difference and set cartesian product respectively.

5) Statements

PLATO has 4 types of statements, print statement, an assignment statement, an if statement and a bare statement.

51) Print statements

A print statement is of the form “PRINT” *expr*“;” where *expr* is any expression. This first evaluates *expr* and then print the resulting value to the console.

5.2) Return statements

A return statement is of the form “RETURN” *expr*“;” where *expr* is any expression. This

evaluates *expr* and then terminate the most recent function call. A function must end with a return statement if and only if it specifies a return type.

5.3) Assignment statements

An assignment statement is of the form *optionalType id* := *expr*; where *optionalType* is either a type or empty, *id* is an identifier and *expr* is any expression. If the type is not empty this is called a declaration and the type given must match the type of the value returned when the expression is evaluated. If the type is empty the identifier must already be declared and the type of the identifier when it was most recently declared must match the type of the value returned when the expression is evaluated.

In an assignment statement *expr* is evaluated and the resulting value is bound to the specified identifier. The identifier will be bound to this value until it is used in another assignment statement or until the end of its scope.

5.3.2) Number assignments

A number can also be assigned by the syntax NUMBER OVER *gId* n*Id* := *expr*; where *gId* is the identifier of a group of extended group, *nId* is any identifier and *expr* is an expression that evaluates to an Integer that is contained in the elements set of *gId*. If a number is not assigned in this fashion it is assumed to be over mathematical the group of integers.

5.3.3) Vector assignments

A matrix can be updated by the syntax *vectorIdentifer*[*indexDefinition*] := *expr*;. The left hand side of the assignment must be the same form as described in section 4.6.1 and *expr* must be an expression which evaluates to a value of the same type as the elements of the indexed vector. This replaces the element at the index whose value is given by *indexDefinition* in the vector with the result of evaluating the expression.

5.4) If statements

An if statement start with the “if” keyword followed by an opening “(“ and closing “)” surrounding an expression called the if condition and then a block of statements called the if body. This may, in turn, be followed by zero or more else if blocks each of which starts with “elseif” followed by an opening “(“ and closing “)” surrounding an expression called the else if condition followed by a block of statements called the else if body. Finally there may be an else block which is an “else” followed block of statements called the else body.

The if condition is evaluated first. If it evaluates to TRUE then the the if body is executed. Otherwise, each else if expression is evaluated in order until one evaluates to TRUE. The else if body corresponding to the else if condition that evaluated to true is then executed. If none of the if or else if conditions hold true and there is an else block then the else body is executed. After this process control flow proceeds to the first line following the all the if, else if and else blocks.

If an if body exists within a function, then all its corresponding else if and else, must match and be consistent in the type that they return, in the case where they do return some value. That is, suppose we have an if block, which returns type INTEGER, then the corresponding else block, must match in the type of its return and thus must also return an INTEGER.

6) Blocks and Scope

6.1) Blocks

A block in PLATO is a sequence of text surrounded by matching “{“ and “}”.
Blocks may be nested, in which case a closing bracket “}” is matched with the most recent preceding opening bracket “{“.

6.2) Scope

Group and function identifiers are global and can be referenced anywhere within the file. Function parameters are local to the function block and cannot be referenced outside that function. Function parameters will mask any group or function identifier with the same name within the scope of that function.
All other identifiers except quantifier identifiers are local to a function. The scope of an identifier is from the line on which it was declared to the end of the function. Local identifiers with the same name as a function parameter or group or function identifier will mask that parameter or identifier within their local scope.

7) PLATO Programs

A PLATO program consists of one or more sections. Each section consists of a header and a block. A PLATO program must have a main section and control flow begins at the first line of the block associated with that section.

7.1) Functions

7.1.1) Function Declaration

A function is a section that consists of a function header followed by a block called the function body. The function header is composed of an optional type, called the return type, followed by an identifier, called function name, and finally a comma separated list of parameter declarations, where a parameter declaration is a type followed by an identifier, in parentheses, called the function parameters. If a return type is specified then the function must provide a code path that will return in a guaranteed fashion. That being said, functions are checked in order to ensure all statements are reachable in some form. If a function contains if, elseif and else blocks, then the blocks are checked to ensure they are consistent in their return type. That is, the return type of

each block matches. In this case the function evaluates to this value and can be used in an assignment or print statement. Otherwise, the function does not evaluate to a value and cannot be used in an assignment or print statement.

Functions are tracked based on the order that they are defined. That is, if your program defines function1, then function2. Then function 1 cannot make reference to function 2. However, function 2 can make reference to function 1, because by the point function 2 is getting defined, function 1 had already been defined and thus tracked. Functions are tracked globally within a file. That is there is a global scope that keeps track of all the function declarations. This in effect, means that the Main function, which is a special function, can “see” all other function declarations in the file.

Every function has its own local scope. This means variables it contains and modifies take place only within the function’s scope and have no global effect.

7.1.2) Function calls

A function is called with the syntax functionIdentifer(expression_1...expression_n). The section before the parentheses must be a function name and the function whose name it is will be referred to as the specified function. The section in parentheses consists of zero or more comma separated expressions, which must match the number of function parameters of the specified function. The expressions will be evaluated in left to right order after which point control will jump to the first line of the function block of the specified function. After the last line of the function control flow will return to immediately after the location where the function was called.

When calling a function, in order for there to be a function match, the name and parameter list types must match.

7.1.3) Main

A main section is the as a function except that its header must be exactly “main()”.

The Main function, which is a special function, can “see” all other function declarations in the file.

7.2) Groups and Extended Groups

PLATO supports groups as well as 2 types of extended groups, rings and fields.

7.1.1) Groups

A group starts with a header that simply says “GROUP” followed by an identifier that is called the group identifier. This is followed by a block called the group body. The group body consists of an assignment statement to a special identifier called “elements” which must be a set of integers. It also contains a function of two variables which must be called “add” which specifies the behavior of the binary operator ‘+’ on any two values in the “elements” set. The set of

elements must form a mathematical group under the '+' operator.

7.1.1) Extended Groups

An extended group has identical syntax to a group except that its header must start with "RING" or "FIELD" instead of "GROUP" and it must also contain a function called "multiply" which specifies the behavior of the binary operator '*' on any two values in the "elements" set. The set of elements must form a ring or field, respectively, under the '+' and '*' operators.

8) PLATO CFG

The following CFG characterizes the syntax of PLATO. **BOLDED** characters represent literal characters that can appear in a PLATO program. Any literal typed in all upper case is also allowed in all lower case in an actual PLATO program. Any space indicates 1 or more required whitespace characters. Regular expression syntax is used to simplify the grammar.

boolean = **TRUE** | **FALSE**
nonZeroDigit = ['**1**'-'**9**']
digit = **0** | nonZeroDigit
number = **0** | (**-**)?nonZeroDigit(digit)*
letter = ['**a**'-'**z**'] | ['**A**'-'**Z**']
alphaNumeric = digit | letter
identifier = letter (alphaNumberic)*
setLiteral = **{}** | **{**expr(**,**exp)***}**
vectorBody = expr **TO** expr (**BY** expr)? | expr(**,**exp)*
vectorLiteral = **[**vectorBody**]**
value = boolean | number | identifier | setLiteral | vectorLiteral
inop = - | **NOT**
binop = **+** | **-** | * | **/** | **%** | ** | **^** | **** | **>** | **>=** | **<** | **<=** | **=** | **!=** | **@** | **AND** | **OR**
groupType = **GROUP** | **RING** | **FIELD**
type = **BOOLEAN | INTEGER | NUMBER** (**OVER** groupType id)? | **SET<**type**>** | **VECTOR<**type**>**
indexer = **(**expr**) | [**expr**]**
casesExpr = { (expr IF expr(;expr IF expr)* ; expr OTHERWISE}
expr = value | **(**expr**)** | id**(**(expr(,expr)*)?**)** | id(indexer)| unop expr | expr binop expr | casesExpr
elseIfBlock = **elseif (**expr**)**statementBlock
elseBlock = **else** statementBlock
statement =**if (**expr**)** statementBlock (elseIfBlock)* (elseBlock)? | **RETURN** expr**;** | **PRINT** expr**;**
| (type)? id **:=** expr**;** | id(indexer) **:=** expr**;** id(indexer);
statementBlock = **{**statement***}**
parameter = id type
parameterList = (param(**,**param)*)?
functionHeader = (type)? id**(**parameterList**)**

functionBlock = functionHeader statementBlock
groupBody = **SET<INTEGER> elements :=** expr**; INTEGER add(INTEGER** id**, INTEGER** id**)**
statementBlock
groupHeader = **GROUP** id
groupBlock = groupHeader**{**groupBody**}**
extendedGroupBody = groupBody type **multiply(INTEGER** id, **INTEGER** id) statementBlock
extendedGroupHeader= **RING** id | **FIELD** id
extendedGroupBlock = extendedGroupHeader**{**extendedGroupBody**}**
generalGroupBlock = groupBlock | extendedGroupBlock
mainBlock = **main()**statementBlock
program = mainBlock (functionBlock)* (generalGroupBlock)*

4. Project Plan

Planning: After we had a list of features from the proposal we started implementing the features 1 by 1. We first got a full pass of the compiler working that took a PLATO program containing a single print statement and compiled to a Java program with a System.out.println. From there we added features one at a time. Every time we added a new feature we also added tests for that feature to the regression test suite. Once the regression tests suite, including the new tests, passed, we considered the new feature complete and moved onto the next one.

Specification and Development:

Given the relatively small size of the team (3 members), the compiler was development using an agile software development approach. That is we started with a compiler composed of a vast array of features that were not necessarily scoped to the project given the time constraints. As a consequence of the agile development approach, we were able to start with a small set of features, incrementally develop and completely sign off on features by ensuring they did not cause any regressions. At this point, we incrementally adopted new features and repeated the latter process.

Testing:

The test plans devised were based on an equivalence partitioning scheme. Given that we are implementing a compiler, each feature falls in a class of its own. Each feature is composed of different classes of smaller features, that must potentially handle input instances of an infinite nature. For instance, consider the feature that allows for the addition of 2 integers. In order to test whether this feature works, that is, given any arbitrary 2 integers, the addition feature works as expected, it suffices to provide a test case where 2 integers are used to validate the addition. Using this test case, provides us with sufficient proof that the addition operation will work for any 2 operands that belong to the class of integers, and thus validates the addition operation on 2 integers, by way of equivalence partitioning. Clearly, in a scenario where one is implementing a compiler, where possible inputs are arbitrary, applying the equivalence partitioning approach to testing proves to be most effective.

Style:

Since the group contained only three people and we frequently used pair programming we did not use an official style guide. In general we tried to follow the guidlines at http://caml.inria.fr/resources/doc/guides/guidelines.en.html although we prefered to put “in” statements and “then” statements on the line following “let” or “if.” We generally tried to use spaces instead of tabs but ince we were using eclipse tabs rendered as spaces so we noticed at the end that we had tabs in the program. We did not have time to fix this formatting issue.

Timeline: Oct 14-Nov 4th: Scanner parser and ast for arithmetic and boolean statements

Nov 5th- Nov 11th: Sast for arithmetic and boolean statements

Nov 18th-Nov 25th: Java Ast and code generation for arithmetic and boolean statements

Nov 25th-Dec 4th: Variables, boolean algebra, type checking

Dec 4th - Dec 20th: Sets, vectors, functions and groups

Roles and responsibilities

Yasser: Functions, if then elseif else, Implementation of testing harness
Daniel: Groups/Rings/Field, Vectorization and logical indexing
Joaquin: Sets, Vectors, Cases expressions

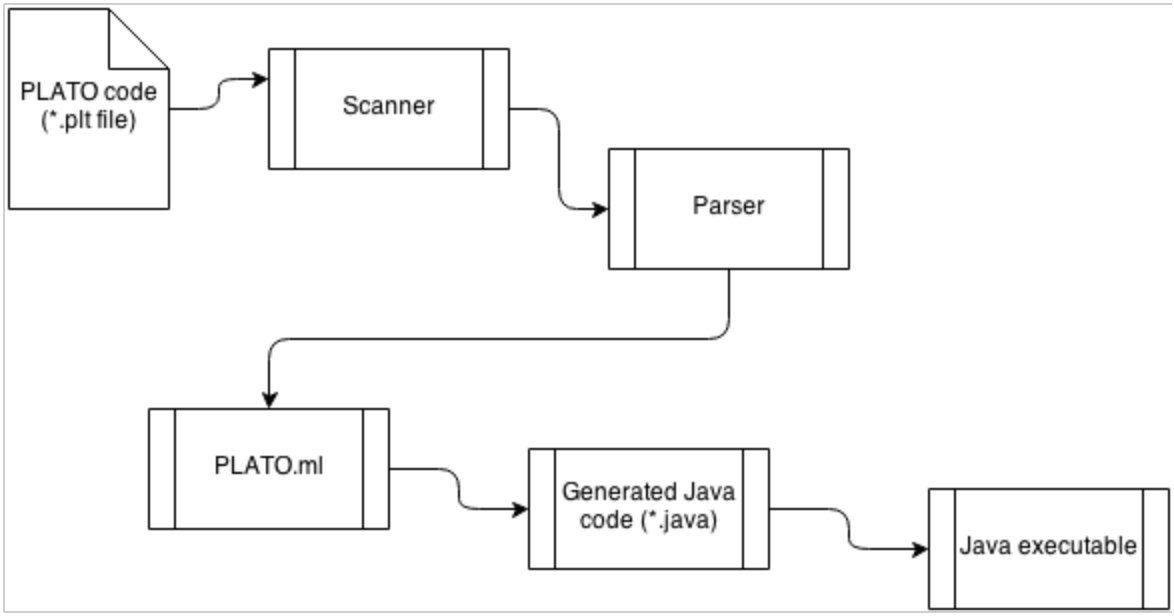
Tools and languages

Ocaml, Shell scripts, Java
Vim, Sublime Text, Eclipse, OcaIde

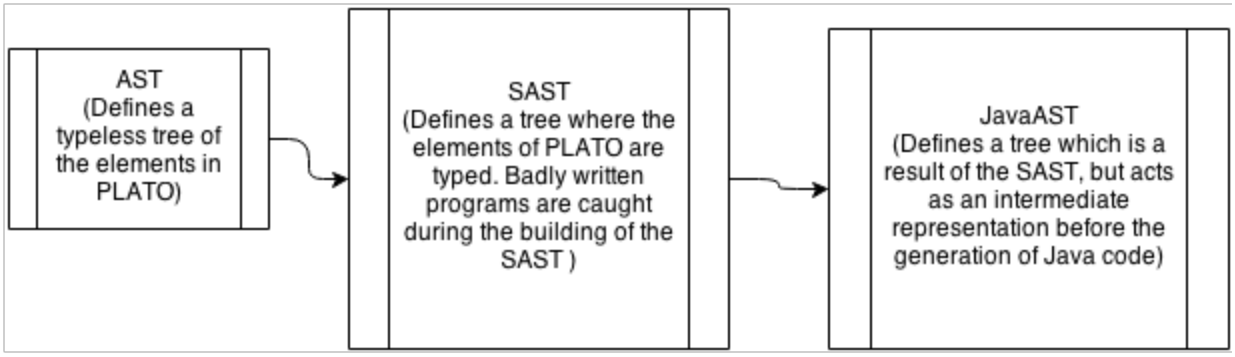
5. Architectural Design

a. Give block diagram showing the major components of your translator

PLATO Code Translation Pipeline:



Data Structures that Capture PLATO Compilation Process:



b. Describe the interfaces between the components
c. State who implemented each component

The scanner produces tokens which are parser by yacc to create the AST. The SAST is created by recursively walking the AST and checking each node for semantic correctness. Group element inverse and validity of group is also checked during this step. The SAST is then

walked to produce the JavaAst. The JavaAst represent the list of java classes. Finally for each class in the JavaAst the tree is walked to generate the code for that java class.

We divided our work up by features so each group member partially implemented each piece of our architecture.

6. Testing Plan

6.1. Representative programs and results

gcdTest.plt

```
main() {
  PRINT gcd(18, 12);
}
INTEGER gcd(INTEGER a, INTEGER b) {
  if (b = 0) {
     RETURN a;
  } else {
     RETURN gcd(b, a % b);
  }
}
```

gcdTest.result

```
6
```

sieveTest.plt

```
main() {
  PRINT primes(100);
}
VECTOR<INTEGER> sieve(VECTOR<INTEGER> v, INTEGER n) {
  if (v[1] ^ 2 <= n) {
     RETURN ([v[1]] @ sieve(v[NOT (v % v[1] = 0)], n));
  } else {
     RETURN v;
  }
}
VECTOR<INTEGER> primes(INTEGER x) {
  RETURN sieve([2 TO x], x);
}
```

sieveTest.result

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

6.2. Test scripts

Regressh.sh

```
#!/bin/bash

tests_dir=$1
failure_count=0
pass_count=0
declare -a regressions
declare -a passes
for source_file in `ls -L $tests_dir/*.plt | xargs -n1 basename`
do
echo "-------------TESTING '$source_file'-----------------"
echo "

source_file_path=$tests_dir"/"$source_file
str_len=${#source_file}
java_out="Main_"${source_file:0:`echo $str_len - 4 | bc`}".java"
compile_result=`./plt.sh $source_file_path dirty no_run`
if [ $? -eq 20 ]
then
src_path_len=${#source_file_path}
expected_out=${source_file_path:0:`echo $src_path_len - 4 | bc`}".result"
if ! [ -e "$expected_out" ]
then
regressions[$failure_count]=$source_file
((failure_count++))
echo "FAILURE $failure_count: '$expected_out' does not exist! Could not find
an expected results file to compare against!"
echo "'$source_file' FAILED regression test!"
if [ -e $java_out ]
then
rm -f $java_out
fi
if [ -e "Main_"${source_file:0:`echo $str_len - 4 | bc`}".class" ]
then
rm -f "Main_"${source_file:0:`echo $str_len - 4 | bc`}".class"
fi
echo -e '\n-------------END-----------------------------'
echo "
continue
fi
```

```
java_out_len=${#java_out}
echo $compile_result > ${java_out:0:`echo $java_out_len - 5 |
bc`}.actual.result
subs_diff_result=`echo $java_out_len - 5 | bc`
diff_result=`diff ${java_out:0:$subs_diff_result}.actual.result $expected_out`

if ! [ $? -eq 0 ]
then
regressions[$failure_count]=$source_file
((failure_count++))
echo "FAILURE $failure_count: Expected output does NOT match actual
output!"
echo "'$source_file' FAILED regression test!"
if [ -e $java_out ]
then
rm -f $java_out
fi
if [ -e "Main_"${source_file:0:`echo $str_len - 4 | bc`}".class" ]
then
rm -f "Main_"${source_file:0:`echo $str_len - 4 | bc`}".class"
fi
echo -e '\n-------------END-----------------------------'
echo "
continue
fi

echo "'$source_file' PASSED regression test!"
if [ -e $java_out ]
then
rm -f $java_out
fi
if [ -e "Main_"${source_file:0:`echo $str_len - 4 | bc`}".class" ]
then
rm -f "Main_"${source_file:0:`echo $str_len - 4 | bc`}".class"
fi
echo -e '\n-------------END-----------------------------'
echo "
passes[$pass_count]=$source_file
((pass_count++))
continue
fi

if ! [ $? -eq 0 ]
```

```
then
regressions[$failure_count]=$source_file
((failure_count++))
echo "FAILURE $failure_count: '$source_file' failed compilation!"
echo "'$source_file' FAILED regression test!"
if [ -e $java_out ]
then
rm -f $java_out
fi
if [ -e "Main_"${source_file:0:`echo $str_len - 4 | bc`}".class" ]
then
rm -f "Main_"${source_file:0:`echo $str_len - 4 | bc`}".class"
fi
echo -e '\n-------------END-----------------------------'
echo "
continue
fi

src_path_len=${#source_file_path}
expected_out=${source_file_path:0:`echo $src_path_len - 4 | bc`}".result"
if ! [ -e "$expected_out" ]
then
regressions[$failure_count]=$source_file
((failure_count++))
echo "FAILURE $failure_count: '$expected_out' does not exist! Could not find
an expected results file to compare against!"
echo "'$source_file' FAILED regression test!"
rm -f $java_out
rm -f "Main_"${source_file:0:`echo $str_len - 4 | bc`}".class"
echo -e '\n-------------END-----------------------------'
echo "
continue
fi

java_out_len=${#java_out}
java ${java_out:0:`echo $java_out_len - 5 | bc`} > ${java_out:0:`echo
$java_out_len - 5 | bc`}.actual.result

subs_diff_result=`echo $java_out_len - 5 | bc`
diff_result=`diff ${java_out:0:$subs_diff_result}.actual.result $expected_out`

if ! [ $? -eq 0 ]
then
```

```
regressions[$failure_count]=$source_file
((failure_count++))
echo "FAILURE $failure_count: Expected output does NOT match actual
output!"
echo "'$source_file' FAILED regression test!"
rm -f $java_out
rm -f "Main_"${source_file:0:`echo $str_len - 4 | bc`}".class"
echo -e '\n-------------END-----------------------------'
echo "
continue
fi

echo "'$source_file' PASSED regression test!"
rm -f $java_out
rm -f "Main_"${source_file:0:`echo $str_len - 4 | bc`}".class"
echo -e '\n-------------END-----------------------------'
echo "
passes[$pass_count]=$source_file
((pass_count++))
done

if [ $failure_count -eq 0 ]
then
echo -e "\n"
echo "------------------------"
echo "ALL PASSED"
echo "0 FAILURES, $pass_count PASSES"
echo "------------------------"
exit
fi

echo -e "\n"
echo "------------------------"
echo "$failure_count FAILURES"
echo "------------------------"

index=0
for failure in "${regressions[@]}"
do
((index++))
echo "FAILURE $index: $failure"
done
```

```
echo "-------------------------"
echo "$pass_count PASSES"
echo "-------------------------"

for pass in "${passes[@]}"
do
echo "$pass"
done
```

PLT.sh

```
#!/bin/bash

source_file_path=$1
source_file=`echo $source_file_path | rev | cut -d "/" -f1 | rev`
str_len=${#source_file}
java_out="Main_"${source_file:0:`echo $str_len - 4 | bc`}".java"
clean_directive=$2

if ! [ -e "$source_file_path" ]
then
echo "Ocaml source file '$source_file_path' does not exist!"
exit 1
fi

compile_result=`./platoc $source_file_path 2>&1`

if ! [ $? -eq 0 ]
then
#echo "'$source_file_path' did NOT translate properly"
echo "TRANSLATION ERROR: $compile_result"
exit 20
fi

echo "'$source_file_path' TRANSLATED successfully"

if ! [ -e "$java_out" ]
then
echo "'$java_out' does not exist! The java code was not generated"
exit 1
```

```
fi

compile_result=`javac $java_out 2>&1`

if ! [ $? -eq 0 ]
then
echo "'$java_out' did NOT compile"
echo "COMPILE ERROR: $compile_result"
if [ $clean_directive == "clean" ]
then
rm -f $java_out
fi
exit 1
fi

echo "'$java_out' COMPILED successfully"
if [ $3 == "run" ]
then
echo -e "\n"
echo "RUNNING..."
java_out_len=${#java_out}
java ${java_out:0:`echo $java_out_len - 5 | bc`}
fi
if [ $clean_directive == "clean" ]
then
rm -f $java_out
rm -f "Main_"${source_file:0:`echo $str_len - 4 | bc`}".class"
fi
exit 0
```

6.3. Test cases and results

advancedArithmeticTest.plt

```
main() {
  PRINT 0 / 1 % 2 ^ 3 + 4;
}
```

advancedArithmeticTest.result

```
4
```

arithmeticTest.plt

```
main() {
```

```
  PRINT 0 + 1 + 2 / 3 - 4 * 5;
}
```

arithmeticTest.result

```
-19
```

booleanAlgebraTest.plt

```
main() {
  PRINT TRUE OR TRUE AND FALSE;
}
```

booleanAlgebraTest.result

```
true
```

booleanATest.plt

```
main() {
  PRINT TRUE;
}
```

booleanTest.result

```
true
```

casesSyntaxTest1.plt

```
main() {
  INTEGER a := 4;
  INTEGER b := -100000;
  PRINT cases(a, b);
  PRINT cases(b, a);
}
INTEGER cases(INTEGER x, INTEGER y) {
  RETURN {
    2          IF 3<2;
    100         IF x<y;
    42          OTHERWISE
  };
}
```

casesSyntaxTest1.result

```
 42
100
```

casesSyntaxTest2.plt

```
main() {
```

```
  PRINT {TRUE IF TRUE; FALSE OTHERWISE};
  PRINT {TRUE IF never(); FALSE OTHERWISE};
}
BOOLEAN never() {
  RETURN FALSE;
}
```

casesSyntaxTest2.result

```
true
false
```

commentTest.plt

```
/*
This is a test for block comments
¬°The compiler will hopefully not be able to read this text!
L/T/Tk at all of th/™se f/•reign ch/£racters
main() {

}
*/
main() {
  PRINT /* another comment */ 5;
}
```

commentTest.result

```
5
```

comparatorTest.plt

```
main() {
  PRINT 1 - 2 + 3 < 4;
  PRINT 1 % 1 <= 2;
  PRINT 0 * 1 >= 2 - 3;
  PRINT 1 > 0;
  PRINT 2 = 2 ^ 1;
}
```

comparatorTest.result

```
true
true
true
true
true
```

doubleMinusTest.plt

```
main() {
  PRINT 1--2;
}
```

doubleMinusTest.Result

```
3
```

extendedGroupTest.plt

```
main() {
  NUMBER OVER RING z3 x := 2;
  PRINT x * x;
  PRINT 2 * 2;
  NUMBER OVER FIELD z5 y := 2;
  NUMBER OVER FIELD z5 y2 := 3;
  PRINT y / y2;
  PRINT 2 / 3;
}
RING z3 {
  SET<INTEGER> elements := {0, 1, 2};
  INTEGER add(INTEGER n1, INTEGER n2) {
    RETURN (n1 + n2) % 3;
  }
  INTEGER multiply(INTEGER n1, INTEGER n2) {
        RETURN (n1 * n2) % 3;
  }
}
FIELD z5 {
  SET<INTEGER> elements := {0, 1, 2, 3, 4};
  INTEGER add(INTEGER n1, INTEGER n2) {
    RETURN (n1 + n2) % 5;
  }
  INTEGER multiply(INTEGER n1, INTEGER n2) {
        RETURN (n1 * n2) % 5;
  }
}
```

extendedGroupTest.result

```
1
4
4
0
```

functionTest1.plt

```
main() {

}
emptyvoidtypelessfunction() {

}
```

functionTest1.result

functionTest2.plt

```
main() {

BOOLEAN x := TRUE;

  NUMBER z2 := 2;
  PRINT x;
PRINT functionwithreturntype();
}
INTEGER functionwithreturntype() {
BOOLEAN y := FALSE;
  y := y OR NOT y;
  INTEGER z1 := 1;
  RETURN -1;
}
```

functionTest2.result

```
true
-1
```

functionTest3.plt

```
main() {

BOOLEAN x := TRUE;

  NUMBER z2 := 2;
  PRINT x;
  PRINT functionWithReturnType();
  PRINT functionWithBooleanReturn();

}
INTEGER functionWithReturnType() {
BOOLEAN y := FALSE;
```

```
  y := y OR NOT y;
  INTEGER z1 := 1;
  RETURN -1;
}
VOID voidFunction() {
PRINT 3;
PRINT 42;
}
BOOLEAN functionWithBooleanReturn() {
RETURN TRUE;
}
```

functionTest3.result

```
true
-1
true
```

groupTest.plt

```
main() {
  NUMBER OVER GROUP z3 x := 2;
  PRINT x + x;
  PRINT 2 + 2;
  PRINT x - x - x;
  PRINT 2 - 2 - 2;
  NUMBER OVER GROUP z3plus1 y := 2;
  PRINT y + y;
}
GROUP z3 {
  SET<INTEGER> elements := {0, 1, 2};
  INTEGER add(INTEGER n1, INTEGER n2) {
        RETURN (n1 + n2) % 3;
  }
}
GROUP z3plus1 {
  SET<INTEGER> elements := {1, 2, 3};
  INTEGER add(INTEGER n1, INTEGER n2) {
        RETURN ((n1 - 1) + (n2 - 1)) % 3 + 1;
  }
}
```

groupTest.result

```
1
4
```

1
-2
3

identifierTest.plt

```
main() {
  BOOLEAN x := TRUE;
  BOOLEAN y := x;
  y := y OR NOT y;
  INTEGER z1 := 1;
  NUMBER z2 := 2;
  PRINT x;
  PRINT y;
  PRINT z1;
  PRINT z2;
}
```

identifierTest.result

```
true
true
1
2
```

ifElseIfElse.plt

```
main() {

BOOLEAN r := TRUE;

  NUMBER z2 := 2;

  if(z2 > 9) {
  PRINT 10;
  }
  elseif(z2 > 7) {
  PRINT 7;
  }
  elseif(z2=2){
  PRINT 3;
  }
  elseif(z2 > 4) {
  PRINT 5;
  }
  elseif(z2 > 6) {
```

```
  PRINT 6;
  }
  elseif(r){
  PRINT 20;
  }
  else {
  PRINT 25;
  }
}
```

ifElseifElse.result
3

ifElseNoElse.plt
```
main() {

BOOLEAN r := TRUE;

  NUMBER z2 := 2;

  if(z2 > 9) {
  PRINT z2;
  }
  elseif(z2 > 3) {
  PRINT 3;
  }
  elseif(r){
  PRINT TRUE;
  }
  elseif(z2<-1){
  PRINT -1;
  }

}
```

ifElseNoElse.result
true

ifElseTest.plt
```
main() {

BOOLEAN r := TRUE;
```

```
  NUMBER z2 := 2;

  if(z2 > 9) {
  PRINT z2;
  }
  else {
  PRINT 42;
  }

}
```

ifElseTest.result

```
42
```

ifTest.plt

```
main() {

BOOLEAN r := TRUE;

  NUMBER z2 := 2;

  if(z2 < 9) {
  PRINT z2;
  }

}
```

ifTest.result

```
2
```

negationTest.plt

```
main() {
  PRINT -1 + 2 / --3;
  PRINT NOT TRUE AND FALSE OR NOT NOT TRUE;
}
```

negationTest.result

```
-1
true
```

nestedIfElseIfTest.plt

```
main() {
```

```
BOOLEAN r := TRUE;

  NUMBER z2 := 2;

  if(z2 < 9) {
  PRINT z2;
         if(r){
         if(z2 > 3) {
       PRINT 3;
         }
         elseif(r){
       PRINT TRUE;
         }
         elseif(z2<-1){
       PRINT -1;
         }
         }
  }


}
```

nestedIfElseIfTest.result

```
2
true
```

printStatementListTest.plt

```
main() {
  PRINT 0;
  PRINT 1;
}
```

printStatementListTest.result

```
0
1
```

printStatementTest.plt

```
main() {
  PRINT 0;
}
```

printStatementTest.result

```
0
```

setTest1.plt

```
main() {
  PRINT {5377357};
}
```

setTest1.result

```
{5377357}
```

setTest2.plt

```
main() {
  SET<INTEGER> mySet := {5377357, 1, 2, 1+2};
  PRINT mySet;
}
```

setTest2.result

```
{5377357, 1, 2, 3}
```

setTest3.plt

```
main() {
  PRINT {1, 2, 3} + {2, 3, 4};
  PRINT {5, 6, 7} ^ {6, 7, 8};
  PRINT {9, 10, 11} \ {10, 11, 12};
  PRINT {{1,2},{2}} + {{2},{1,2,3}};
}
```

setTest3.result

```
{1, 2, 3, 4}
{6, 7}
{9}
{{2}, {1, 2}, {1, 2, 3}}
```

setTest4.plt

```
main() {
  PRINT {{1}, 1, 10};
}
```

setTest4.result

```
TRANSLATION ERROR: Fatal error: exception PLATO.PlatoError("Set has
heterogeneous types: Set<Integers>, Integers, Integers")
```

vectorTest1.plt

```
main() {
```

```
  PRINT [5377357];
}
```

vectorTest1.result

```
[5377357]
```

vectorTest2.plt

```
main() {
  VECTOR<INTEGER> myVector := [5377357, 1, 2, 1+2];
  PRINT myVector;
}
```

vectorTest2.result

```
[5377357, 1, 2, 3]
```

vectorTest3.plt

```
main() {
  VECTOR<INTEGER> myVector := [5, 4, 3, 2, 1];
  myVector[2] := 9000;
  VECTOR<BOOLEAN> myBoolVector := [TRUE, FALSE, FALSE, TRUE, TRUE];
  myBoolVector[5] := FALSE;
  PRINT myVector;
  PRINT myBoolVector;
}
```

vectorTest3.result

```
[5, 9000, 3, 2, 1]
[true, false, false, true, false]
```

vectorTest4.plt

```
main() {
  VECTOR<INTEGER> myVector := [5, 4, 3, 2, 1];
  myVector[2] := 9000;
  myVector[4] := myVector[2]^2;
  PRINT myVector;
}
```

vectorTest4.result

```
main() {
  VECTOR<INTEGER> myVector := [5, 4, 3, 2, 1];
  myVector[2] := 9000;
```

```
  myVector[4] := myVector[2]^2;
  PRINT myVector;
}
```

vectorTest5.plt

```
main() {
  VECTOR<INTEGER> myVector := [1 TO 5 BY 2];
  myVector[1] := 9000;
  myVector[2] := myVector[1]^2;
  PRINT myVector;
}
```

vectorTest5.result

```
[9000, 81000000, 5]
```

vectorTest6.plt

```
main() {
  VECTOR<INTEGER> myVector := [1 TO 5 BY 2];
  VECTOR<INTEGER> myVector2 := myVector[[TRUE, FALSE, TRUE]];
  PRINT myVector2;
}
```

vectorTest6.result

```
[1, 5]
```

vectorTest7.plt

```
main() {
  VECTOR<BOOLEAN> myVector := NOT [TRUE, FALSE, TRUE];
  VECTOR<INTEGER> myVector2 := - [1, 2, 3];
  VECTOR<INTEGER> myVector3 := [1, 2, 3] + 1;
  VECTOR<INTEGER> myVector4 := [1, 2, 3] - 1;
  VECTOR<INTEGER> myVector5 := [1, 2, 3] * 1;
  VECTOR<INTEGER> myVector6 := [1, 2, 3] / 1;
  VECTOR<INTEGER> myVector7 := [1, 2, 3] % 1;
  VECTOR<INTEGER> myVector8 := [1, 2, 3] ^ 1;
  VECTOR<BOOLEAN> myVector9 := [1, 2, 3] > 1;
  VECTOR<BOOLEAN> myVector10 := [1, 2, 3] >= 1;
  VECTOR<BOOLEAN> myVector11 := [1, 2, 3] < 1;
  VECTOR<BOOLEAN> myVector12 := [1, 2, 3] <= 1;
  VECTOR<BOOLEAN> myVector13 := [1, 2, 3] = 1;
  VECTOR<INTEGER> myVector14 := 1 + [1, 2, 3];
  PRINT myVector;
  PRINT myVector2;
```

```
  PRINT myVector3;
  PRINT myVector4;
  PRINT myVector5;
  PRINT myVector6;
  PRINT myVector7;
  PRINT myVector8;
  PRINT myVector9;
  PRINT myVector10;
  PRINT myVector11;
  PRINT myVector12;
  PRINT myVector13;
  PRINT myVector14;
}
```

vectorTest7.result

```
[false, true, false]
[-1, -2, -3]
[2, 3, 4]
[0, 1, 2]
[1, 2, 3]
[1, 2, 3]
[0, 0, 0]
[1, 2, 3]
[false, true, true]
[true, true, true]
[false, false, false]
[true, false, false]
[true, false, false]
[2, 3, 4]
```

vectorTest8.plt

```
main() {
  PRINT [0] @ [0];
  PRINT 0 @ [0];
  PRINT [0] @ 0;
}
```

vectorTest8.result

```
[0, 0]
[0, 0]
[0, 0]
```

casesOfDifferentTypes.plt

```
main() {
  INTEGER a := 4;
  INTEGER b := -100000;
  PRINT cases(a, b);
  PRINT cases(b, a);
}
INTEGER cases(INTEGER x, INTEGER y) {
  RETURN {
    2          IF 3<2;
    FALSE       IF x<y;
    42         OTHERWISE
  };
}
```

casesOfDifferentTypes.result

```
TRANSLATION ERROR: Fatal error: exception PLATO.PlatoError("Cases literal has
inconsistent types")
```

casesWithNonBooleanConditionals.plt

```
main() {
  PRINT {TRUE IF 45; FALSE OTHERWISE};
  PRINT {TRUE IF never(); FALSE OTHERWISE};
}
BOOLEAN never() {
  RETURN FALSE;
}
```

casesWithNonBooleanConditionals.result

```
TRANSLATION ERROR: Fatal error: exception PLATO.PlatoError("Cases literal has
inconsistent types")
```

castExceptionTest.plt

```
main() {
  BOOLEAN x := 0;
}
```

castExceptionTest.result

```
TRANSLATION ERROR: Fatal error: exception PLATO.PlatoError("Cannot cast from Integers
to Booleans")
```

functionParameterListTypMismatchTest.plt

```
main() {
```

```
BOOLEAN r := TRUE;

  NUMBER z2 := 2;
  PRINT functionWithIfElse(TRUE, 0, FALSE, 9);
  PRINT functionWithBooleanReturn();

}
INTEGER functionWithIfElse(BOOLEAN x, INTEGER a, INTEGER b, BOOLEAN c) {
  if(x) {
        RETURN -1;
  }
  elseif(3 > 0) {
        RETURN 4;
  }
  else {
        RETURN 0;
  }
}
BOOLEAN functionWithBooleanReturn() {
RETURN TRUE;
}
```

functionParameterListTypMismatchTest.result

```
TRANSLATION ERROR: Fatal error: exception PLATO.PlatoError("Function call:
functionWithIfElse with the list of provided of parameters could not be matched to any
existing function.")
```

garbageFileTest.plt

```
TRANSLATION ERROR: Fatal error: exception PLATO.PlatoError("Function call:
functionWithIfElse with the list of provided of parameters could not be matched to any
existing function.")
```

garbageFiltTest.result

```
TRANSLATION ERROR: Fatal error: exception Parsing.Parse_error
```

groupTimesTest.plt

```
main() {
  NUMBER OVER GROUP z3 x := 2;
  PRINT x * x;
}
GROUP z3 {
  SET<INTEGER> elements := {0, 1, 2};
  INTEGER add(INTEGER n1, INTEGER n2) {
```

```
        RETURN (n1 + n2) % 3;
  }
}
```

groupTimesTest.result

```
TRANSLATION ERROR: Fatal error: exception PLATO.PlatoError("Cannot apply times to types
z3, z3")
```

invalidOperatorExceptionTest.plt

```
main() {
  INTEGER x := 0 + TRUE;
}
```

invalidOperatorExceptionTest.result

```
TRANSLATION ERROR: Fatal error: exception PLATO.PlatoError("Cannot apply plus to types
Integers, Booleans")
```

noidentityGroupTest.plt

```
main() {
  NUMBER OVER GROUP z3 x := 2;
  PRINT x + x;
  PRINT 2 + 2;
  PRINT x - x - x;
  PRINT 2 - 2 - 2;
}
GROUP z3 {
  SET<INTEGER> elements := {0, 1, 2};
  INTEGER add(INTEGER n1, INTEGER n2) {
    RETURN 0;
  }
}
```

noidentityGroupTest.result

```
TRANSLATION ERROR: Fatal error: exception PLATO.PlatoError("Error while generating
group, ring or field z3. Could not find identity element")
```

noinverseGroupTest.plt

```
main() {
  NUMBER OVER GROUP z4 x := 2;
  PRINT x + x;
  PRINT 2 + 2;
  PRINT x - x - x;
  PRINT 2 - 2 - 2;
```

```
}
GROUP z4 {
  SET<INTEGER> elements := {0, 1, 2};
  INTEGER add(INTEGER n1, INTEGER n2) {
    RETURN (n1 * n2) % 4;
  }
}
```

noinverseGroupTest.result

TRANSLATION ERROR: Fatal error: exception PLATO.PlatoError("Error while generating group, ring or field z4. Could not find inverse of 0")

nonBooleanIfExprTest.plt

```
main() {

BOOLEAN r := TRUE;

  NUMBER z2 := 2;

  if(z2) {
  PRINT z2;
          if(r){
          if(z2 > 3) {
        PRINT 3;
          }
          elseif(r){
        PRINT TRUE;
          }
          elseif(z2<-1){
        PRINT -1;
          }
          }
  }


}
```

nonBooleanIfExprTest.result

TRANSLATION ERROR: Fatal error: exception PLATO.PlatoError("if statements expect an expression of type boolean. Actual type of expression: Integers")

noReturn.plt

```
main() {

BOOLEAN r := TRUE;

  NUMBER z2 := 2;
  PRINT functionWithIfElse(TRUE);
  PRINT functionWithBooleanReturn();

}
INTEGER functionWithIfElse(BOOLEAN x) {
BOOLEAN y := x;
  y := y OR NOT y;
  INTEGER z1 := 1;
  PRINT x;
  if(z1 > 3){
  PRINT 0;
  }
  elseif (y){
  RETURN 42;
  }
}
BOOLEAN functionWithBooleanReturn() {
RETURN TRUE;
}
```

noReturn.result

```
TRANSLATION ERROR: Fatal error: exception PLATO.PlatoError("Return statement
incompatible types for the Function: functionWithIfElse. Required: Integers. Found: void")
```

notFoundFunctionTest.plt

```
main() {

BOOLEAN r := TRUE;

  NUMBER z2 := 2;
  PRINT randomFunctionCall();

}
INTEGER functionWithIfElse(BOOLEAN x) {
BOOLEAN y := x;
  y := y OR NOT y;
  INTEGER z1 := 1;
  PRINT x;
```

```
  if(z1 > 3){
  PRINT 0;
  }
  elseif (y){
  RETURN 42;
  }
  RETURN -1;
}
BOOLEAN functionWithBooleanReturn() {
RETURN TRUE;
}
```

notFoundFunctionTest.result

```
TRANSLATION ERROR: Fatal error: exception PLATO.PlatoError("Function:
randomFunctionCall NOT_FOUND_EXCEPTION")
```

redeclaredidentifierTest.plt

```
main() {
  INTEGER x := 1;
  INTEGER x := 2;
  PRINT x;
}
```

redeclaredidentifierTest.result

```
TRANSLATION ERROR: Fatal error: exception PLATO.PlatoError("Identifier x is already
declared")
```

ringDivideTest.plt

```
main() {
  NUMBER OVER RING z3 x := 2;
  PRINT x / x;
}
RING z3 {
  SET<INTEGER> elements := {0, 1, 2};
  INTEGER add(INTEGER n1, INTEGER n2) {
        RETURN (n1 + n2) % 3;
  }
  INTEGER multiply(INTEGER n1, INTEGER n2) {
        RETURN (n1 * n2) % 3;
  }
}
```

ringDivideTest.result

TRANSLATION ERROR: Fatal error: exception PLATO.PlatoError("Cannot apply divide to types z3, z3")

setOfDifferentTypes.plt

```
/*
{x} and {y} are of different types, so we shouldn't be able to union set {x} with set {y}
*/
main() {
  NUMBER OVER GROUP z3 x := 2;
  NUMBER OVER GROUP z2 y := 1;
  PRINT {x}+{y};
}
GROUP z2 {
  SET<INTEGER> elements := {0, 1};
  INTEGER add(INTEGER n1, INTEGER n2) {
        RETURN (n1 + n2) % 2;
  }
}
GROUP z3 {
  SET<INTEGER> elements := {0, 1, 2};
  INTEGER add(INTEGER n1, INTEGER n2) {
    RETURN (n1 + n2) % 3;
  }
}
```

setOfDifferentTypes.result

TRANSLATION ERROR: Fatal error: exception PLATO.PlatoError("Cannot apply plus to types Set<z3>, Set<z2>")

unclosedGroupTest.plt

```
main() {
  NUMBER OVER GROUP z3 x := 2;
  PRINT x + x;
  PRINT 2 + 2;
  PRINT x - x - x;
  PRINT 2 - 2 - 2;
}
GROUP z3 {
  SET<INTEGER> elements := {0, 1, 2};
  INTEGER add(INTEGER n1, INTEGER n2) {
    RETURN n1 + n2;
  }
}
```

unclosedGroupTest.result
TRANSLATION ERROR: Fatal error: exception PLATO.PlatoError("Group addition must be closed and associative")

undeclaredidentifierTest.plt
```
main() {
  PRINT x;
}
```

undeclaredidentifierTest.result
TRANSLATION ERROR: Fatal error: exception PLATO.PlatoError("Undeclared identifier x")

unreachableCodeTest.plt
```
main() {

BOOLEAN r := TRUE;

  NUMBER z2 := 2;
  PRINT functionWithIfElse(TRUE);
  PRINT functionWithBooleanReturn();

}
INTEGER functionWithIfElse(BOOLEAN x) {
  if(x) {
        RETURN -1;
  }
  elseif(3 > 0) {
        RETURN 4;
  }
  else {
        RETURN 0;
  }
RETURN -2;
}
BOOLEAN functionWithBooleanReturn() {
RETURN TRUE;
}
```

unreachableCodeTest.result
TRANSLATION ERROR: Fatal error: exception PLATO.PlatoError("Function has unreachable code!")

variableCallTest.plt

```
main() {
  INTEGER x := 0;
  x();
}
```

6.4. Test case purposes

i. Void function test - Testing void function support
ii. Typed function test - Testing support for a function with a return type
iii. Multiple functions within a file. - Testing support of multiple functions, with different names and return types. Testing support for calling all different functions from the main.
iv. Calling multiple functions in a file - Testing both that function calls work, and that all functions are being tracked from the scope of the main function.
v. Calling only one function in a file with multiple functions - Testing that everything works, and all functions are being tracked, even though only one of the functions are being called.
vi. Unreachable code in a function - Testing error is shown when a function contains unreachable statements.
vii. Calling a function that exists with a list of parameters that do not match the type of parameters that were defined in the existing function's header - Testing an error is shown when a user calls a function with the incorrect parameter types.
viii. Calling a function which is not defined - Testing error is shown when a function is called but it does not actually exist.
ix. Calling a function f2 from a function f1, where f1 is defined before f2 - testing to make sure error is shown if a function makes reference to another function which has not been defined as of yet.
x. If elseif and else test - Testing basic functionality of what the LRM claims support for in a given if body that is chained with a series of elseif's and an else.
xi. If elseif but no else test - Testing that the compiler supports if blocks which do not contain an else block.
xii. If else test - Testing support for no elseif chain, for a given if body.
xiii. If test - Testing support for an if body with no elseif chain, and no else statement.
xiv. Nested if elseifs and else test - Combining tests x, xi, xii, and xiii while including multiple layers of nesting within a given if body, in order to test for support of nesting, for validly defined if bodies.
xv. Else with no if test - Testing exception is thrown the compiler does not support the generation of an else block that is not associated with an if block.
xvi. Elseif with no if test - Testing error is shown the compiler does not support the generation of an elseif block that is not associated with an if block.

xvii. An if(expression) where expression does not evaluate to a boolean type test - Testing error is shown because every expression inside an if(expression) or elseif(expression) must evaluate to a boolean type, otherwise, the if body is not valid.

xviii. A chain of if elseif else blocks with an inconsistent set of return statements - Testing that an error is shown, if an if block is not consistent throughout in the type that is being returned, within a function that returns some value.

xix. A simple print statement to test printing

xx. A list of print statements to test statement lists

xxi. An arithmetic expression to test the arithmetic operators

xxii. An arithmetic expression with exponents and modulo to test more advanced arithmetic operators

xxiii. Adding two boolean to test operator type checking

xxiv. Using comparators to test comparator type checking

xxv. Printing a boolean to test booleans

xxvi. Printing a boolean algebra expression to test boolean algebra

xxvii. Using a double minus to test parsing of double negation

xxviii. Use of negation and not to test unary operators

xxix. Printing an identifier to test variable assignment

xxx. Printing an identifier before it is defined to test identifier scopting

xxxi. Declaring an identifier twice to test scoping

xxxii. Creating a file with random characters to test parsing

xxxiii. Assigning an integer to a boolean to test casting

xxxiv. Declaring a group and addition and subtracting numbers to test group generation

xxxv. Declaring a group where addition is unclosed to test group verification

xxxvi. Declaring a group where addition is not assiative to test group verification

xxxvii. Declaring a group where additive inverses do not exist to test group verification

xxxviii. Multiplying on a group to test groups versus rings

xxxix. Declaring a ring and field and testing addition and multiplication to test ring and field generation

xl. Dividing an element over a ring to test rings versus fields

xli. Printing a set with a single element

xlii. Printing a set with multiple elements

xliii. Printing a set that contained expressions as its elements

xliv. Trying to add elements of disparate types to sets

xlv. Printing a vector with a single element

xlvi. Printing a vector with multiple elements

xlvii. Printing a vector of booleans

xlviii. Assigning to a vector

xlix. Logical indexing of a vector
l. Vectorization of operators
li. Appending vectors
lii. Using the cases syntax for cases inside a function
liii. Using the cases syntax for inline cases
liv. Using the cases syntax with inconsistent types should throw an error
lv. Using the cases syntax with non-boolean conditionals should throw an error

6.5. Description of testing process

lvi. A testing harness was implemented in bash script, that automated the process of running every test, automatically comparing it with the expect result, and reporting on PASS/FAIL status. The testing suite only required a directory which contains all the tests, and it automated the process.
lvii. The testing harness would then provide a summary report of the passes/failures of running the tests in some given directory, along with log files that keep track of what took place when the tests were run as well as the results.
lviii. This testing harness made it really easy for us to test whether the implementation of a new feature, or the fixing of a bug, resulted in any regressions. The reason for this is because the testing harness would run all the regression tests and report on them.

Yasser:
Implemented test harness/suite
Functions test cases (i - ix)
If elseif and else test cases (x - xviii)
Daniel:
Basic feature test cases (xviii - xxxiii)
If elseif and else test cases (xxxiv - xl)
Joaquín:
Set test cases (xli - xliv)
Vector test cases (xlv - li)
Cases syntax test cases (lii-lv)

To perform testing, we use a shell script that uses our compiler on every test file in a specified folder and compares the result with a result file of the same name in the same folder.

7. Lessons Learned

d. Each team member should explain his or her most important learning
    i. Yasser:
        1. Once your ocaml code is compiled, everything just simply and magically works as you expect it to. Unless of course you wrote code which aimed to introduce bugs.
        2. The only way to learn the implementation of a compiler is to actually do it. Understanding the process of writing a compiler, as well as its theoretical inner-workings, is completely different from actually implementing the compiler.
        3. I definitely learned to love ocaml, as I found myself many times asking how one could actually implement many of the features we did, without the expressibility of ocaml itself.
        4. Definitely, start as early as possible. As soon as you learn how to use ocamlyacc, start playing with different features in your compiler, so as to step away from the theoretical realm of compilers, and get a real sense for its implementation.
    ii. Daniel:
        1. If you try to do too much at once this can get very complicated but if you break each feature down into simple steps it's actually not that hard to implement complicated features.
        2. Regression testing and logging doesn't seem that important at first but it becomes really critical when you start adding more features
        3. Testing this that shouldn't compile is just as important as testing things that should
    iii. Joaquin:
        1. Creating sample programs for a feature of your compiler before implementing it helps you catch issues before they happen
        2. For a project this big, it is best to start with the most basic functionality possible and build up on that. Communication and version control are key when people edit the same files simultaneously
        3. Programming in a language you have created can be very rewarding
e. Include any advice the team has for future teams
    i. Simply put, start early, and learn to love OCaml.
    ii. Make sure to scope down the compiler's features while in the phase of proposing your compiler idea. Your proposal and features you include

in it, should be conservative, but exceed what is sufficient to claim you implemented a compiler.

8. Appendix

f. Attach a complete code listing of your translator with each module signed by its author
g. Do not include any ANTLR-generated files, only the .g sources.

We do not include authors because every file was touch by all 3 of us

Ast.mli

```
type operator =
        | At
        | Not
        | And
        | Or
        | Negation
        | Plus
        | Minus
        | Times
        | Divide
        | Mod
        | Raise
        | LessThan
        | LessThanOrEqual
        | GreaterThan
        | GreaterThanOrEqual
        | Equal
        | SetDifference
        | VectorAccess

type platoType =
        | BooleanType
        | NumberType of string * string
        | SetLiteralType of platoType
        | VectorLiteralType of platoType
        | CasesLiteralType of platoType
        | NeutralType

type platoQuantifier =
        | WhichQuantifier
        | SomeQuantifier
```

```
        | AllQuantifier

type platoFunctionType =
        | VoidType
        | OtherType of platoType

type expression =
        | Boolean of bool
        | Number of int
        | Identifier of string
        | Unop of operator * expression
        | Binop of operator * expression * expression
        | SetLiteral of expression list
        | FunctionCall of string * expression list
        | VectorLiteral of expression list
        | VectorRange of expression * expression * expression
        | CasesLiteral of (expression * expression) list * expression

type statement =
        | VoidCall of expression
        | Print of expression
        | Return of expression
        | If of expression * statementBlock * elseIfBlock list * elseBlock
        | IfNoElse of expression * statementBlock * elseIfBlock list
  | Assignment of string * expression
  | VectorAssignment of string * expression * expression
        | Declaration of platoType * string * expression
and statementBlock =
          StatementBlock of statement list
and elseBlock =
        ElseBlock of statementBlock
and elseIfBlock =
        ElseIfBlock of expression * statementBlock

type parameter = Parameter of platoType * string

type functionHeader = {
   returnType : platoFunctionType;
   functionName : string;
   parameters : parameter list;
  }

type functionBlock =
```

```
        FunctionDeclaration of functionHeader * statementBlock

type mainBlock =
        MainBlock of statementBlock

type groupHeader =
        GroupHeader of string

type groupBody =
        GroupBody of expression * functionBlock

type extendedGroupHeader =
        | RingHeader of string
        | FieldHeader of string

type extendedGroupBody =
        ExtendedGroupBody of groupBody * functionBlock


type extendedGroupBlock =
        | GroupDeclaration of groupHeader * groupBody
  | ExtendedGroupDeclaration of extendedGroupHeader * extendedGroupBody

type program  =
        Program of mainBlock * functionBlock list * extendedGroupBlock list
```

Sast.mli

```
open Ast;;

type  variableDeclaration =
        string * Ast.platoType
type functionDeclaration =
                string * Ast.platoFunctionType * Ast.parameter list

type typedExpression =
        | TypedBoolean of bool * Ast.platoType
        | TypedNumber of int * Ast.platoType
  | TypedIdentifier of string  * Ast.platoType
        | TypedUnop of operator * Ast.platoType * typedExpression
        | TypedBinop of operator * Ast.platoType * typedExpression * typedExpression
        | TypedSet of Ast.platoType * typedExpression list
        | TypedVector of Ast.platoType * typedExpression list
```

```
        | TypedVectorRange of Ast.platoType * typedExpression * typedExpression *
typedExpression
        | TypedCases of Ast.platoType * (typedExpression * typedExpression) list *
typedExpression
        | TypedFunctionCall of Ast.platoFunctionType * string * typedExpression list

type typedStatement =
        | TypedVoidCall of typedExpression
        | TypedPrint of typedExpression
        | TypedReturn of Ast.platoFunctionType * typedExpression
        | TypedIf of Ast.platoFunctionType * typedExpression * typedStatementBlock *
typedElseIfBlock list * typedElseBlock
        | TypedIfNoElse of typedExpression * typedStatementBlock * typedElseIfBlock list
        | TypedAssignment of variableDeclaration * typedExpression
        | TypedVectorAssignment of variableDeclaration * typedExpression * typedExpression
        | TypedDeclaration of variableDeclaration * typedExpression
and typedElseIfBlock =
          TypedElseIfBlock of typedExpression * typedStatementBlock
and typedElseBlock =
          TypedElseBlock of typedStatementBlock
and typedStatementBlock =
          TypedStatementBlock of typedStatement list

type typedParameter =
          TypedParameter of variableDeclaration

type typedFunctionBlock =
          TypedFunctionDeclaration of Ast.functionHeader * typedStatementBlock

type typedMainBlock =
          TypedMainBlock of typedStatementBlock


type typedExtendedGroupBlock =
        | TypedGroupDeclaration of string * int list * int list list * int list
        | TypedRingDeclaration of string * int list * int list list * int listint list list
  | TypedFieldDeclaration of string * int list * int list list * int list       * int list list * int list * int

type typedProgram =
          TypedProgram of typedMainBlock * typedFunctionBlock list * typedExtendedGroupBlock
list
```

JavaAst.mli

```
open Sast;;

type javaType =
        | JavaBooleanType
        | JavaIntType
        | JavaSetLiteralType
        | JavaVectorLiteralType
        | JavaCasesLiteralType
        | JavaNeutralType

type javaPrimitive =
        | JavaBoolean of bool
        | JavaInt of int

type javaValue =
        | JavaValue of javaPrimitive
        | JavaMap of string * string list * string list

type javaExpression =
        | JavaConstant of javaValue
        | JavaVariable of string
        | JavaReturn of javaExpression
        | JavaIf of javaExpression * javaBlock * javaElseIf list * javaElse
        | JavaIfNoElse of javaExpression * javaBlock * javaElseIf list
        | JavaAssignment of string * javaExpression
        | JavaVectorAssignment of string * javaExpression * javaExpression
        | JavaDeclaration of javaType * string * javaExpression option
        | JavaCall of string * string * javaExpression list
        | JavaTernaryChain of (javaExpression * javaExpression) list * javaExpression
and javaElseIf =
        JavaElseIf of javaExpression * javaBlock
and javaElse =
        JavaElse of javaBlock
and javaStatement =
         JavaStatement of javaExpression
and javaBlock =
         JavaBlock of javaStatement list

(* type javaFunctionHeader =
         JavaFunctionHeader of javaFunctionType * string*)
```

```
type javaMethod        =
          | JavaMain of javaBlock
                | JavaDefaultConstructor of string * javaBlock
          | JavaFunction of Ast.functionHeader * javaBlock

type javaClass =
          JavaClass of string * string * javaMethod list

type javaClassList =
          JavaClassList of javaClass list
```

Scanner.mll

```
{ open Parser open Logger}

rule token = parse
  | [' ' '\t' '\r' '\n'] { token lexbuf }
  | "/*"     { comment lexbuf }
          | "BOOLEAN" { BOOLEAN_TYPE }
          | "INTEGER" { INTEGER_TYPE }
          | "NUMBER" { NUMBER_TYPE }
          | "SET"   { SET_TYPE }
          | "VOID" { VOID_TYPE }
          | "VECTOR"   { VECTOR_TYPE }
          | "TO"   { VECTOR_TO }
          | "BY"   { VECTOR_BY }
          | "WHICH" { WHICH_QUANTIFIER }
          | "SOME" { SOME_QUANTIFIER }
          | "ALL" { ALL_QUANTIFIER }
  | "TRUE" { BOOLEAN(true) }
          | "FALSE" { BOOLEAN(false) }
          | "NOT" { NOT }
          | "OR" { OR }
          | "AND" { AND }
          | "OVER" { OVER }
          | "PRINT" { PRINT }
          | "RETURN" { RETURN }
          | "GROUP" { GROUP }
          | "RING" { RING }
          | "FIELD" { FIELD }
          | "elements" { ELEMENTS }
          | "add" { ADD }
          | "multiply" { MULTIPLY }
```

```
          | "IF"|"if" { IF }
          | "ELSEIF"|"elseif" { ELSEIF }
          | "ELSE"|"else" { ELSE }
          | "OTHERWISE" { OTHERWISE }
          | "main()" { MAIN_HEADER }
          | '@' { AT }
          | '+' { PLUS }
          | '-' { MINUS }
          | '\\' { BACKSLASH }
          | '*' { TIMES }
          | '/' { DIVIDE }
          | '%' { PERCENT }
          | '^' { CARET }
          | '<' { LESS_THAN }
          | '>' { GREATER_THAN }
          | '=' { EQUAL }
          | ':' { COLON }
          | ',' { COMMA }
  | ';' { SEMICOLON }
          | '{' { OPEN_BRACE }
          | '}' { CLOSE_BRACE }
          | '[' { OPEN_BRACKET }
          | ']' { CLOSE_BRACKET }
          | '(' { LPAREN }
          | ')' { RPAREN }
          | '0' { NUMBER(0) }
  | ['1'-'9']['0'-'9']* as number { NUMBER(int_of_string number) }
          | ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9']* as identifier { IDENTIFIER(identifier) }
  | eof { EOF }
  | _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

 and comment = parse
  "*/" { token lexbuf }
| _    { comment lexbuf }
```

Parser.mly

```
%{ open Ast open Logger %}

%token BOOLEAN_TYPE INTEGER_TYPE NUMBER_TYPE SET_TYPE VECTOR_TYPE
VOID_TYPE
%token VECTOR_TO VECTOR_BY
```

```
%token WHICH_QUANTIFIER SOME_QUANTIFIER ALL_QUANTIFIER
%token NOT NEGATION
%token LESS_THAN GREATER_THAN EQUAL
%token AT PLUS MINUS BACKSLASH TIMES DIVIDE PERCENT CARET AND OR
%token OVER PRINT RETURN GROUP RING FIELD ELEMENTS ADD MULTIPLY
%token COLON COMMA SEMICOLON LPAREN RPAREN OPEN_BRACE CLOSE_BRACE
MAIN_HEADER EOF OPEN_BRACKET CLOSE_BRACKET IF ELSEIF ELSE OTHERWISE
%token <bool> BOOLEAN
%token <int> NUMBER
%token <string> IDENTIFIER

%nonassoc VECTOR_TO VECTOR_BY
%left OR
%left AND
%left EQUAL
%left LESS_THAN LESS_THAN_OR_EQUAL GREATER_THAN
GREATER_THAN_OR_EQUAL
%left PLUS MINUS BACKSLASH AT
%left TIMES DIVIDE PERCENT
%nonassoc NOT NEGATION
%right CARET
%left LPAREN RPAREN

%start program
%type <Ast.program> program

%%

platoType:
  | BOOLEAN_TYPE { BooleanType }
          | INTEGER_TYPE  { NumberType("field", "Integers") }
          | NUMBER_TYPE  { NumberType("field", "Integers") }
          | NUMBER_TYPE OVER GROUP IDENTIFIER { NumberType("group", $4) }
          | NUMBER_TYPE OVER RING IDENTIFIER { NumberType("ring", $4) }
          | NUMBER_TYPE OVER FIELD IDENTIFIER { NumberType("field", $4) }
          | SET_TYPE LESS_THAN platoType GREATER_THAN  { SetLiteralType($3) }
          | VECTOR_TYPE LESS_THAN platoType GREATER_THAN  { VectorLiteralType($3) }

platoFunctionType:
          | VOID_TYPE { VoidType }
  | platoType { OtherType($1) }

commaSeparatedExpression:
```

```
        { [] }
        | expression { [$1] }
        | commaSeparatedExpression COMMA expression { $3::$1 }

commaSeparatedExpressionNonemptyList:
        expression { [$1] }
        | commaSeparatedExpressionNonemptyList COMMA expression { $3::$1 }

setLiteral:
        OPEN_BRACE CLOSE_BRACE {SetLiteral([])}
        | OPEN_BRACE commaSeparatedExpressionNonemptyList CLOSE_BRACE
{SetLiteral(List.rev $2)}

vectorLiteral:
        OPEN_BRACKET CLOSE_BRACKET {VectorLiteral([])}
        | OPEN_BRACKET commaSeparatedExpressionNonemptyList CLOSE_BRACKET
{VectorLiteral(List.rev $2)}

quantifier:
        | SOME_QUANTIFIER {SomeQuantifier}
        | ALL_QUANTIFIER {AllQuantifier}
/*
        WHICH_QUANTIFIER {WhichQuantifier}
*/
semicolonSeparatedCaseExpressionNonemptyList:
        expression IF expression { [($1, $3)] }
        | semicolonSeparatedCaseExpressionNonemptyList SEMICOLON expression IF
expression { ($3, $5)::$1 }

casesLiteral:
         OPEN_BRACE semicolonSeparatedCaseExpressionNonemptyList SEMICOLON
expression OTHERWISE CLOSE_BRACE { CasesLiteral(List.rev $2, $4) }

expression:
        | BOOLEAN { Boolean($1) }
        |       NUMBER { Number($1) }
        | IDENTIFIER { Identifier($1) }
        | IDENTIFIER OPEN_BRACKET expression CLOSE_BRACKET { Binop(VectorAccess,
Identifier($1), $3) }
        | NOT expression { Unop(Not, $2) }
        | MINUS expression %prec NEGATION      { Unop(Negation, $2) }
        | expression OR expression { Binop(Or, $1, $3) }
        | expression AND expression { Binop(And, $1, $3) }
```

```
        | expression PLUS expression { Binop(Plus, $1, $3) }
        | expression MINUS expression { Binop(Minus, $1, $3) }
        | expression BACKSLASH expression { Binop(SetDifference, $1, $3) }
        | expression TIMES expression { Binop(Times, $1, $3) }
        | expression DIVIDE expression { Binop(Divide, $1, $3) }
        | expression PERCENT expression { Binop(Mod, $1, $3) }
        | expression CARET expression { Binop(Raise, $1, $3) }
        | expression LESS_THAN expression { Binop(LessThan, $1, $3) }
        | expression LESS_THAN EQUAL expression %prec LESS_THAN_OR_EQUAL {
Binop(LessThanOrEqual, $1, $4) }
        | expression GREATER_THAN expression { Binop(GreaterThan, $1, $3) }
        | expression GREATER_THAN EQUAL expression %prec
GREATER_THAN_OR_EQUAL { Binop(GreaterThanOrEqual, $1, $4) }
        | expression EQUAL expression { Binop(Equal, $1, $3) }
        | setLiteral { $1 }
        | LPAREN expression RPAREN { $2 }
        | IDENTIFIER LPAREN commaSeparatedExpression RPAREN { FunctionCall($1,
List.rev $3) }
        | vectorLiteral { $1 }
        | expression AT expression { Binop(At, $1, $3) }
        /*
        | quantifier IDENTIFIER IN vectorLiteral SATISFIES expr {QuantifierLiteral($1,
Identifier($2), $4, $6)}
        */
        | OPEN_BRACKET expression VECTOR_TO expression CLOSE_BRACKET {
VectorRange($2, $4, Number(1)) }
        | OPEN_BRACKET expression VECTOR_TO expression VECTOR_BY expression
CLOSE_BRACKET { VectorRange($2, $4, $6) }
        | casesLiteral { $1 }

statement:
  | expression SEMICOLON { VoidCall($1) }
  | PRINT expression SEMICOLON { Print($2) }
  | RETURN expression SEMICOLON { Return($2) }
  | IF LPAREN expression RPAREN statementBlock elseIfBlockList elseBlock { If($3, $5, List.rev
$6, $7) }
  | IF LPAREN expression RPAREN statementBlock elseIfBlockList { IfNoElse($3, $5, List.rev $6)
}
  | IDENTIFIER COLON EQUAL expression SEMICOLON { Assignment($1, $4) }
  | IDENTIFIER OPEN_BRACKET expression CLOSE_BRACKET COLON EQUAL expression
SEMICOLON { VectorAssignment($1, $3, $7) }
  |       platoType IDENTIFIER COLON EQUAL expression SEMICOLON { Declaration($1, $2,
$5) }
```

```
statementList:
  |/* empty */ { [] }
  | statementList statement { $2::$1 }

statementBlock:
    OPEN_BRACE statementList CLOSE_BRACE { StatementBlock(List.rev $2) }

elseBlock:
        ELSE statementBlock { ElseBlock($2) }

elseIfBlock:
        ELSEIF LPAREN expression RPAREN statementBlock { ElseIfBlock($3, $5) }

elseIfBlockList:
        | { [] }
        | elseIfBlockList elseIfBlock { $2::$1 }

parameter:
        platoType IDENTIFIER { Parameter($1, $2) }

parameterWithComma:
  | parameter COMMA parameter { [$3; $1]}
        | parameterWithComma COMMA parameter { $3::$1 }

parameterList:
        | { [] }
        | parameter { [$1] }
        | parameterWithComma { $1 }

addFunctionHeader:  INTEGER_TYPE ADD LPAREN INTEGER_TYPE IDENTIFIER COMMA
INTEGER_TYPE IDENTIFIER RPAREN { { returnType = OtherType(NumberType("field",
"Integers"));

         functionName = "add";

         parameters = [Parameter(NumberType("field", "Integers"), $5);
Parameter(NumberType("field", "Integers"), $8)]  } }


addFunctionBlock: addFunctionHeader statementBlock { FunctionDeclaration($1, $2) }
```

```
multiplyFunctionHeader:  INTEGER_TYPE MULTIPLY LPAREN INTEGER_TYPE IDENTIFIER
COMMA INTEGER_TYPE IDENTIFIER RPAREN { { returnType =
OtherType(NumberType("field", "Integers"));

            functionName = "multiply";

            parameters = [Parameter(NumberType("field", "Integers"), $5);
Parameter(NumberType("field", "Integers"), $8)]  } }


multiplyFunctionBlock: multiplyFunctionHeader statementBlock { FunctionDeclaration($1, $2) }

functionHeader:
  | platoFunctionType IDENTIFIER LPAREN parameterList RPAREN { { returnType = $1;

            functionName = $2;

            parameters = List.rev $4 } }
           | IDENTIFIER LPAREN parameterList RPAREN { { returnType = VoidType;

functionName = $1;

parameters = List.rev $3 } }


functionBlock:
             functionHeader statementBlock { FunctionDeclaration($1, $2) }

functionBlockList:
  | { [] }
  | functionBlockList functionBlock { $2::$1 }

mainBlock:
    MAIN_HEADER statementBlock { MainBlock($2) }

groupHeader:
    GROUP IDENTIFIER { GroupHeader($2) }

groupBody:
     SET_TYPE LESS_THAN INTEGER_TYPE GREATER_THAN ELEMENTS COLON EQUAL
setLiteral SEMICOLON addFunctionBlock { GroupBody($8, $10) }

groupBlock:
```

```
   groupHeader OPEN_BRACE groupBody CLOSE_BRACE { GroupDeclaration($1, $3) }

extendedGroupHeader:
  | RING IDENTIFIER { RingHeader($2) }
          | FIELD IDENTIFIER { FieldHeader($2) }

extendedGroupBody:
    groupBody multiplyFunctionBlock { ExtendedGroupBody($1, $2) }

extendedGroupBlock:
  | groupHeader OPEN_BRACE groupBody CLOSE_BRACE { GroupDeclaration($1, $3) }
  | extendedGroupHeader OPEN_BRACE extendedGroupBody CLOSE_BRACE {
ExtendedGroupDeclaration($1, $3) }

groupBlockList:
  | { [] }
          | groupBlockList extendedGroupBlock { $2 :: $1 }

program:
   mainBlock functionBlockList groupBlockList { Program($1, List.rev $2, List.rev $3) }
```

Logger.ml

```
open Printf;;
open Ast;;
open Sast;;
open JavaAst;;

let operatorToString = function
          | At -> "at"
          | Not ->  "not"
          | And -> "and"
          | Or -> "or"
          | Negation -> "negation"
          | Plus -> "plus"
          | Minus -> "minus"
          | Times -> "times"
          | Divide -> "divide"
          | Mod -> "mod"
          | Raise -> "raise"
          | LessThan -> "lessThan"
          | LessThanOrEqual -> "lessThanOrEqual"
          | GreaterThan -> "greaterThan"
```

```
        | GreaterThanOrEqual -> "greaterThanOrEqual"
        | Equal ->  "equal"
        | SetDifference ->  "setDifference"
        | VectorAccess -> "vectorAccess"

let rec typeToString = function
        | BooleanType -> "Booleans"
        | NumberType(extendeGroupType, groupName) -> "Number over " ^ extendeGroupType ^
groupName
        | SetLiteralType(subtype) -> "Set of " ^ (typeToString subtype)
        | VectorLiteralType(subtype) -> "Vector of " ^ (typeToString subtype)
        | CasesLiteralType(subtype) -> "Set of Cases of type " ^ (typeToString subtype)
        | NeutralType -> "Neutral Type"

let functionTypeToString = function
        | VoidType -> "void";
        | OtherType(platoType) -> typeToString platoType

let logToFile mode permissions newline fileName logString =
        let fileHandle = open_out_gen mode permissions fileName
        in (if newline
                then fprintf fileHandle "%s\n"  logString
                else fprintf fileHandle "%s" logString);
                close_out fileHandle

let logToFileAppend = logToFile [Open_creat; Open_append] 0o777

let logToFileOverwrite = logToFile [Open_creat; Open_wronly] 0o777

(* Logging for PLATO AST *)
let logListToAst logStringList =
        (logToFileAppend true) "Ast.log" (String.concat " " logStringList)

let logStringToAst logString =
        logListToAst [logString]

let logOperatorAst operator = logStringToAst (operatorToString operator)

let rec logPlatoTypeAst = function
        | BooleanType -> logStringToAst "BooleanType"
        | NumberType(extendeGroupType, groupName) -> logListToAst ["Number Type over
group "; extendeGroupType; groupName]
```

```
        | SetLiteralType(subType) -> ignore (logListToAst ["SetLiteral Type of subtype "]);
logPlatoTypeAst subType
        | VectorLiteralType(subType) -> ignore (logListToAst ["VectorLiteral Type of subtype "]);
logPlatoTypeAst subType
        | CasesLiteralType(subType) -> ignore (logListToAst ["CasesLiteral Type of subtype "]);
logPlatoTypeAst subType
        | NeutralType -> logStringToAst "NeutralType"

let rec logExpressionAst = function
        | Boolean(booleanValue) -> logListToAst ["Boolean"; string_of_bool booleanValue]
        | Number(integerValue) -> logListToAst ["Number"; string_of_int integerValue]
        | Identifier(identifierName) -> logListToAst ["Identifier"; identifierName]
        | Unop(operator, expression) -> logOperatorAst operator; logExpressionAst expression
        | Binop(operator, expression1, expression2) -> logOperatorAst operator;
logExpressionAst expression1; logExpressionAst expression2
        | SetLiteral(expressionList) ->
                logListToAst ["Set of"; string_of_int (List.length expressionList);"elements"];
                ignore (List.map logExpressionAst expressionList)
        | VectorLiteral(expressionList) ->
                logListToAst ["Vector of"; string_of_int (List.length expressionList);"elements"];
                ignore (List.map logExpressionAst expressionList)
        | VectorRange(fromExpression, toExpression, byExpression) ->
                logStringToAst "Vector range from ";
   logExpressionAst fromExpression;
          logStringToAst " to ";
   logExpressionAst toExpression;
          logStringToAst " by ";
                logExpressionAst byExpression
        | CasesLiteral(cases, defaultCase) -> logStringToAst "Cases Literal "
        | FunctionCall(functionName, expressionList) ->
                logListToAst ["Call to"; functionName; "with parameters"];
                ignore (List.map logExpressionAst expressionList)

let rec logStatementAst = function
        | VoidCall(voidFunction) ->
                logStringToAst "Void call to";
                logExpressionAst voidFunction
        | Print(printValue) ->
                logStringToAst "Print";
                logExpressionAst(printValue)
        | Return(expression) ->
                logStringToAst "Return";
                logExpressionAst(expression)
```

```
        | If(predicate, ifBody, elseIfBlocks, elseBlock) ->
          logListToAst ["If statement with"; string_of_int (List.length elseIfBlocks); "else if blocks
and an else block"];
                logExpressionAst predicate;
                logStatementBlockAst ifBody;
                ignore (List.map logElseIfBlockAst elseIfBlocks);
                logElseBlockAst elseBlock
        | IfNoElse(predicate, ifBody, elseIfBlocks) ->
          logListToAst ["If statement with"; string_of_int (List.length elseIfBlocks); "else if blocks"];
                logExpressionAst predicate;
                logStatementBlockAst ifBody;
                ignore (List.map logElseIfBlockAst elseIfBlocks)
        | Assignment(identifier, rhs) ->
                logListToAst ["Assignment of identifier"; identifier; "to"];
                logExpressionAst rhs
        | VectorAssignment(identifier, indexer, rhs) ->
                logListToAst ["Vector assignment of identifier"; identifier];
                logStringToAst "with indexer";
                logExpressionAst indexer;
                logStringToAst "to";
                logExpressionAst rhs
        | Declaration(platoType, identifier, rhs) ->
                logStringToAst "Declaration"; logPlatoTypeAst platoType;
                logListToAst ["Identifier"; identifier]; logExpressionAst(rhs)
and logStatementBlockAst = function
          StatementBlock(statementList) -> logListToAst ["StatementBlock of size"; string_of_int
(List.length statementList)]; ignore (List.map logStatementAst statementList)
and logElseIfBlockAst =  function
        | ElseIfBlock(predicate, elseIfBody) ->
                logExpressionAst predicate;
                logStatementBlockAst elseIfBody;
and logElseBlockAst = function
        | ElseBlock(elseBody) ->
                logStatementBlockAst elseBody

let logMainBlockAst = function
          MainBlock(statementBlock) -> logStringToAst "MainBlock"; logStatementBlockAst
statementBlock

let logParameterAst = function
        | Parameter(parameterType, parameterName) -> logListToAst ["parameter";
parameterName; "of type"; typeToString parameterType]
```

```
let logFunctionHeaderAst functionHeader =
	logListToAst ["Plato function with name"; functionHeader.functionName; "return type";
functionTypeToString functionHeader.returnType];
	ignore (List.map logParameterAst functionHeader.parameters)

let logFunctionBlockAst = function
	| FunctionDeclaration(functionHeader, statementBlock) ->
		logFunctionHeaderAst functionHeader;
		logStatementBlockAst statementBlock

let logGroupHeaderAst = function
	| GroupHeader(groupName) ->	logListToAst ["Group with name "; groupName]

let logExtendedGroupHeaderAst = function
	| RingHeader(groupName) ->logListToAst ["Ring with name "; groupName]
	| FieldHeader(groupName) ->logListToAst ["Field with name "; groupName]

let logGroupBodyAst = function
	| GroupBody(elements, addFunctionBlock) ->
		logStringToAst "Elements ";
		logExpressionAst elements;
		logFunctionBlockAst addFunctionBlock

let logExtendedGroupBodyAst = function
	| ExtendedGroupBody(GroupBody(elements, addFunctionBlock), multiplyFunctionBlock)
->
		logStringToAst "Elements ";
		logExpressionAst elements;
		logFunctionBlockAst addFunctionBlock;
		logFunctionBlockAst multiplyFunctionBlock

let logGroupBlockAst = function
	| GroupDeclaration(groupHeader, groupBody) ->
		logGroupHeaderAst groupHeader;
		logGroupBodyAst groupBody
	| ExtendedGroupDeclaration(extendedGroupHeader, extendedGroupBody) ->
		logExtendedGroupHeaderAst extendedGroupHeader;
		logExtendedGroupBodyAst extendedGroupBody

let logProgramAst = function
	  Program(mainBlock, functionBlockList, groupBlockList) ->
			logListToAst ["Program of size"; "1"];
			logMainBlockAst mainBlock;
```

```
                    ignore (List.map logFunctionBlockAst functionBlockList);
                    ignore (List.map logGroupBlockAst groupBlockList)

(* Logging for PLATO SAST *)
let rec typeToString = function
        | BooleanType -> "Booleans"
        | NumberType(_, groupName) -> groupName
        | SetLiteralType(platoType) -> ("Set<" ^ (typeToString platoType) ^ ">")
        | VectorLiteralType(platoType) -> ("Vector<" ^ (typeToString platoType) ^ ">")
        | CasesLiteralType(platoType) -> ("Cases<" ^ (typeToString platoType) ^ ">")
        | NeutralType -> "NeutralTypes"

let logListToSast logStringList =
        (logToFileAppend true) "Sast.log" (String.concat " " logStringList)

let logStringToSast logString =
        logListToSast [logString]

let logOperatorSast operator = logStringToSast (operatorToString operator)

let logPlatoTypeSast = function
        | BooleanType -> logStringToSast "Boolean Type"
        | NumberType(extendedGroupType, groupName) -> logListToSast ["Number Type over
group"; extendedGroupType; groupName]
        | SetLiteralType(platoType) -> logStringToSast ("SetLiterals<" ^ (typeToString platoType)
^ ">")
        | VectorLiteralType(platoType) -> logStringToSast ("VectorLiterals<" ^ (typeToString
platoType) ^ ">")
        | CasesLiteralType(platoType) -> logStringToSast ("CasesLiterals<" ^ (typeToString
platoType) ^ ">")
        | NeutralType -> logStringToSast "Neutral Type"

let rec logExpressionSast = function
        | TypedBoolean(booleanValue, _) -> logListToSast ["Boolean"; string_of_bool
booleanValue]
        | TypedNumber(numberValue, numberType) -> logListToSast ["Number"; string_of_int
numberValue; " over "; typeToString numberType]
  | TypedIdentifier(variableName, variableType) -> logListToSast ["Variable";  variableName; "of
type"; typeToString variableType]
        | TypedUnop(unaryOperator, operatorType, operatorExpression) ->
                logStringToSast "unary operator";
                logOperatorSast unaryOperator;
                logListToSast ["of type";  typeToString operatorType];
```

```
                logStringToSast "acting on";
                logExpressionSast operatorExpression;
        | TypedBinop(binaryOperator, operatorType, operatorExpression1, operatorExpression2)
->
                logStringToSast "binary operator";
                logListToSast ["of type";  typeToString operatorType];
                logOperatorSast binaryOperator;
                logStringToSast "acting on";
                logExpressionSast operatorExpression1;
                logStringToSast "and acting on";
                logExpressionSast operatorExpression2
        | TypedSet(platoType, expressionList) ->
                logStringToSast "set literal";
                logListToSast ["of type";  typeToString platoType];
                logStringToSast "containing the expressions";
                ignore (List.map logExpressionSast expressionList)
        | TypedVector(platoType, expressionList) ->
                logStringToSast "vector literal";
                logListToSast ["of type";  typeToString platoType];
                logStringToSast "containing the expressions";
                ignore (List.map logExpressionSast expressionList)
        | TypedVectorRange(vectorType, fromExpression, toExpression, byExpression) ->
                logStringToSast "vector range ";
                logListToSast ["of type";  typeToString vectorType];
          logStringToSast " from ";
   logExpressionSast fromExpression;
          logStringToSast " to ";
   logExpressionSast toExpression;
          logStringToSast " by ";
                logExpressionSast byExpression
        | TypedCases(pltType, _, _) ->
          logStringToSast " Cases of type ";
          logStringToSast (typeToString pltType)
        | TypedFunctionCall(functionType, functionName, typedExpressionList) ->
                logListToSast ["Call to"; functionName; "of type"; functionTypeToString
functionType; "with parameters"];
                ignore (List.map logExpressionSast typedExpressionList)

let rec logStatementSast = function
        | TypedVoidCall(voidFunction) ->
                logStringToSast "Void call to";
                logExpressionSast voidFunction
        | TypedPrint(printExpression) ->
```

```
                    logStringToSast "Print";
                    logExpressionSast(printExpression)
        | TypedReturn(returnType, returnExpression) ->
                    logListToSast ["Return of typ"; functionTypeToString returnType];
                    logExpressionSast(returnExpression)
        | TypedIf(returnType, predicate, ifBody, elseIfBlocks, elseBlock) ->
          logListToSast ["If statement with"; string_of_int (List.length elseIfBlocks); "else if blocks
and an else block and return type"; functionTypeToString returnType];
                logExpressionSast predicate;
                logStatementBlockSast ifBody;
                ignore (List.map logElseIfBlockSast elseIfBlocks);
                logElseBlockSast elseBlock
        | TypedIfNoElse(predicate, ifBody, elseIfBlocks) ->
          logListToSast ["If statement with"; string_of_int (List.length elseIfBlocks); "else if
blocks"];
                logExpressionSast predicate;
                logStatementBlockSast ifBody;
                ignore (List.map logElseIfBlockSast elseIfBlocks);
        | TypedAssignment((variableName, variableType), newValue) ->
                logListToSast ["Assign"; variableName; "of type"];
                logPlatoTypeAst variableType;
                logStringToSast " to value ";
                logExpressionSast(newValue)
        | TypedVectorAssignment((identifier, vectorType), indexer, rhs) ->
                logListToSast ["Vector assignment of identifier"; identifier; "of type"; typeToString
vectorType];
                logStringToSast "with indexer";
                logExpressionSast indexer;
                logStringToSast "to";
                logExpressionSast rhs
        | TypedDeclaration((variableName, variableType), newValue) ->
          logListToSast ["Declare"; variableName;  "as type"];
                logPlatoTypeSast variableType;
                logStringToSast "and assign to value ";
                logExpressionSast(newValue)
and logStatementBlockSast = function
          TypedStatementBlock(statementList) -> logListToSast ["StatementBlock of size";
string_of_int (List.length statementList)]; ignore (List.map logStatementSast statementList)
and logElseIfBlockSast =  function
        | TypedElseIfBlock(predicate, elseIfBody) ->
                logExpressionSast predicate;
                logStatementBlockSast elseIfBody
and logElseBlockSast = function
```

```
        | TypedElseBlock(elseBody) ->
                logStatementBlockSast elseBody

let logMainBlockSast = function
          TypedMainBlock(statementBlock) -> logStringToSast "MainBlock";
logStatementBlockSast statementBlock

let logParameterSast = function
        | Parameter(parameterType, parameterName) -> logListToSast ["parameter";
parameterName; "of type"; typeToString parameterType]

let logFunctionHeaderSast functionHeader =
        logListToSast ["Plato function with name"; functionHeader.functionName; "return type";
functionTypeToString functionHeader.returnType];
        ignore (List.map logParameterSast functionHeader.parameters)

let logFunctionBlockSast = function
        | TypedFunctionDeclaration(functionHeader, statementBlock) ->
                logFunctionHeaderSast functionHeader;
                logStatementBlockSast statementBlock

let logTableSast tableName table =
        logStringToSast ("Table " ^ tableName);
        ignore (List.map (fun intList -> logListToSast (List.map string_of_int intList)) table)

let logGroupBlockSast = function
        | TypedGroupDeclaration(groupName, groupElements, additionTable, additiveInverseList)
->
                logListToSast ["Group with name "; groupName; "and elements"];
                logListToSast (List.map string_of_int groupElements);
                logTableSast "additionTable" additionTable;
                logListToSast ("additiveInverseList"::(List.map string_of_int additiveInverseList))
        | TypedRingDeclaration(ringName, ringElements, additionTable, additiveInverseList,
multiplicationTable) ->
                logListToSast ["Ring with name "; ringName; "and elements"];
                logListToSast (List.map string_of_int ringElements);
                logTableSast "additionTable" additionTable;
                logListToSast ("additiveInverseList"::(List.map string_of_int additiveInverseList));
                logTableSast "multiplicationTable" multiplicationTable;
  | TypedFieldDeclaration(fieldName, fieldElements, additionTable, additiveInverseList,
multiplicationTable, multiplicitiveInverseList, additiveIdentity) ->
                logListToSast ["Field with name "; fieldName; "and elements"];
                logListToSast (List.map string_of_int fieldElements);
```

```
                logTableSast "additionTable" additionTable;
                logListToSast ("additiveInverseList"::(List.map string_of_int additiveInverseList));
                logTableSast "multiplicationTable" multiplicationTable;
                logListToSast ("multiplicitiveInverseList"::(List.map string_of_int
multiplicitiveInverseList));
                logListToSast ["Additive identity"; string_of_int additiveIdentity]

let logProgramSast = function
   TypedProgram(mainBlock, typedFunctionBlockList, typedGroupBlockList) ->
                        logListToSast ["Program of size"; "1"];
                        logMainBlockSast mainBlock;
                 ignore (List.map logFunctionBlockSast typedFunctionBlockList);
                        ignore (List.map logGroupBlockSast typedGroupBlockList)

(* Logging for Java AST *)
let logListToJavaAst logStringList =
        (logToFileAppend true) "JavaAst.log" (String.concat " " logStringList)

let logStringToJavaAst logString =
        logListToJavaAst [logString]

let logjavaPrimitiveAst = function
        | JavaBoolean(booleanValue) -> logListToJavaAst ["Java boolean"; string_of_bool
booleanValue]
  | JavaInt(intValue) -> logListToJavaAst ["Java int"; string_of_int intValue]

let rec logJavaValueAst = function
        | JavaValue(javaPrimitive) -> logjavaPrimitiveAst javaPrimitive
        | JavaMap(mapName, keyList, valueList) ->
                logListToJavaAst ["Java map "; mapName; "with keys"];
                logListToJavaAst keyList;
          logStringToJavaAst " and values ";
          logListToJavaAst valueList

let rec logJavaExpressionAst = function
        | JavaConstant(javaValue) -> logJavaValueAst javaValue
        | JavaVariable(stringValue) -> logListToJavaAst ["Java variable"; stringValue]
        | JavaReturn(expressionToReturn) ->
                logListToJavaAst ["Return statement"];
                logJavaExpressionAst expressionToReturn
        | JavaAssignment(variableName, variableValue) ->
                logListToJavaAst ["Java assignment of variable"; variableName; "to"];
                logJavaExpressionAst variableValue
```

```
        | JavaDeclaration(variableType, variableName, variableValue) ->
          (logListToJavaAst ["Java assignment of variable"; variableName; "assigned to"];
                match variableValue with
                   Some(javaExpressionValue) -> logJavaExpressionAst javaExpressionValue
            | None -> () (* do nothing *))
        | JavaCall(className, methodName, methodParameters) ->
                logListToJavaAst ["Java call to"; className; "."; methodName; "with";
string_of_int (List.length methodParameters); "parameters"];
                ignore (List.map logJavaExpressionAst methodParameters)
        | _ -> ()

let logJavaStatementAst = function
   JavaStatement(javaExpression) -> logStringToJavaAst "Java bare statement";
logJavaExpressionAst javaExpression

let logJavaBlockAst = function
          JavaBlock(statementList) -> logListToJavaAst ["Java block with"; string_of_int
(List.length statementList); "statements"]; ignore (List.map logJavaStatementAst statementList)

let logJavaFunctionHeaderAst functionHeader =
        logListToAst ["Function with name"; functionHeader.functionName; "return type";
functionTypeToString functionHeader.returnType];
        ignore (List.map logParameterAst functionHeader.parameters)

let logJavaMethodAst = function
          | JavaMain(methodBlock) ->
                        logStringToJavaAst "Java main";
                        logJavaBlockAst methodBlock
          | JavaDefaultConstructor(className, constructorBody) ->
                        logListToJavaAst ["Constuctor for Java class"; className];
                        logJavaBlockAst constructorBody
                | JavaFunction(methodHeader, methodBlock) ->
                        logJavaFunctionHeaderAst methodHeader;
                        logJavaBlockAst methodBlock

let logJavaClassAst = function
          JavaClass(className, superClassName, javaMethodList) -> logListToJavaAst ["Java
class "; className; "extending"; superClassName; "with"; string_of_int (List.length
javaMethodList); "methods"];
                ignore (List.map logJavaMethodAst javaMethodList)

let logJavaClassListAst = function
```

```
    JavaClassList(javaClassList) -> logListToJavaAst ["Java class list of size"; string_of_int
(List.length javaClassList)]; ignore (List.map logJavaClassAst javaClassList)
```

PLATO.ml

```
open Ast;;
open Logger;;
open Sast;;
open JavaAst;;
open PlatoLibraryStrings;;
open Filename;;

exception PlatoError of string
exception DebugException of string

let allTrue list = List.fold_left (&&) true list

let undeclaredVariableException variableName = PlatoError("Undeclared identifier " ^
variableName)

let redeclaredVariableException variableName =
        PlatoError("Identifier " ^ variableName ^ " is already declared")

let castException expressionType  variableType =
        PlatoError("Cannot cast from " ^ (typeToString expressionType) ^ " to " ^ (typeToString
variableType))

let operatorException operator inputTypeList =
        PlatoError("Cannot apply " ^ (operatorToString operator) ^ " to types " ^ (String.concat ", "
(List.map typeToString inputTypeList)))

let identityException groupName =
        PlatoError("Error while generating group, ring or field " ^ groupName ^ ".  Could not find
identity element")

let inverseException groupName element=
        PlatoError("Error while generating group, ring or field " ^ groupName ^ ".  Could not find
inverse of " ^ (string_of_int element))

let voidFunctionHasReturnException functionName = PlatoError("Function: " ^ functionName ^ "
is a void function. Void functions cannot return an expression")
```

```
let missingReturnStmtException functionName functionReturnType = PlatoError("Function: " ^ functionName ^ " is a typed function of type " ^ functionReturnType ^ ". Missing return statement. Expecting return statement of type " ^ functionReturnType ^ ".")

let incompatibleTypesReturnStmt functionName functionReturnType actualReturn  = PlatoError("Return statement incompatible types for the Function: " ^ functionName ^ ". Required: " ^ functionReturnType ^ ". Found: " ^ actualReturn)

let noneBooleanExpressionInElseIf expressionType = PlatoError("elseif statements expect an expression of type boolean. Actual type of expression: " ^ (typeToString expressionType))
let noneBooleanExpressionInIf expressionType = PlatoError("if statements expect an expression of type boolean. Actual type of expression: " ^ (typeToString expressionType))
let unreachableCodeException = PlatoError("Function has unreachable code!")

let functionCallParameterTypeMismatchException functionName = PlatoError("Function call: " ^ functionName ^ " with the list of provided of parameters could not be matched to any existing function.")

let functionNotFoundException functionName = PlatoError("Function: " ^ functionName ^ " NOT_FOUND_EXCEPTION")
let heterogeneousSetLiteralException variableTypes =
        PlatoError("Set has heterogeneous types: "^(String.concat ", " (List.map typeToString variableTypes)))

let heterogeneousVectorLiteralException variableTypes =
        PlatoError("Vector has heterogeneous types: "^(String.concat ", " (List.map typeToString variableTypes)))

(* Intepreter for simple statements *)
let evaluateSimpleUnop unopValue = function
        | Negation -> -unopValue
        | _ -> raise(PlatoError("Invalid simple unop"))

let evaluateSimpleBinop binopValue1 binopValue2 = function
  | Plus -> binopValue1 + binopValue2
        | Minus -> binopValue1 - binopValue2
        | Times -> binopValue1 * binopValue2
        | Divide -> binopValue1 / binopValue2
        | Mod -> binopValue1 mod binopValue2
        | Raise -> int_of_float ((float_of_int binopValue1) ** (float_of_int binopValue2))
        | _ -> raise(PlatoError("Invalid simple binop"))

let rec evaluateSimpleExpression identifierName1 identifierName2 input1 input2 = function
```

```
        | Number(numberValue) -> numberValue
        | Identifier(variableName) ->
                (if variableName = identifierName1 then input1
                 else if variableName = identifierName2 then input2
                 else raise(undeclaredVariableException variableName))
        | Unop(unaryOperator, unopExpression) -> evaluateSimpleUnop
(evaluateSimpleExpression identifierName1 identifierName2 input1 input2 unopExpression)
unaryOperator
        | Binop(binaryOperator, binopExpression1, binopExpression2) -> evaluateSimpleBinop
(evaluateSimpleExpression identifierName1 identifierName2 input1 input2 binopExpression1)
(evaluateSimpleExpression identifierName1 identifierName2 input1 input2 binopExpression2)
binaryOperator
        | _ -> raise(PlatoError("Expressions in groups', rings' or fields' add or multiply can only
contain basic arithmetic operators"))

let evaluateSimpleSet = function
        | SetLiteral(expressionList) -> List.map (evaluateSimpleExpression "" "" 0 0)
expressionList
        | _ -> raise(PlatoError("A group, ring or field must have a set of elements"))

let evaluateSimpleStatement identifierName1 identifierName2 input1 input2 = function
        | Return(javaExpression) ->
                evaluateSimpleExpression identifierName1 identifierName2 input1 input2
javaExpression
        | _ -> raise(PlatoError("Statements in groups', rings' or fields' add or multiply can only be
returns"))

let evaluateSimpleBinaryFunction input1 input2 = function
        | FunctionDeclaration({ returnType = OtherType(NumberType("field", "Integers"));
functionName = _; parameters = [Parameter(NumberType("field", "Integers"), identifierName1);
Parameter(NumberType("field", "Integers"), identifierName2)]}, StatementBlock([javaStatement]))
->
                evaluateSimpleStatement identifierName1 identifierName2 input1 input2
javaStatement
        | _ -> raise(PlatoError("Functions in groups, rings or fields can only be add or multiply"))

(* Convert Ast to Sast *)
let extractPltTypeFromFuncType = function
        OtherType(pltType) -> pltType
        | _ -> raise(DebugException("If this point was reached, cannot have voidType"))

let canCast fromType toType =
  if fromType = toType
```

```
        then true
        else
                match toType with
                | NumberType(_, _) -> (fromType = NumberType("field", "Integers"))
                | _ -> false

let canFunctionCast funcType1 funcType2 =
        match funcType1 with
                VoidType -> funcType2=VoidType
                | OtherType(pltType) -> canCast pltType (extractPltTypeFromFuncType
funcType2)

type symbolTable = {
  mutable variables : variableDeclaration list;
}

type translationEnvironemnt = {
        scope : symbolTable;
}

type functionsTable = {
        mutable functionsList : functionDeclaration list;
}
type globalEnvironment = {
        globalScope : functionsTable;
}

let rec findVariable scope variableName =
  let finderFunction = (function (name, _) -> name = variableName)
        in List.find finderFunction scope.variables

let updateScope scope variableDeclaration =
        let (variableName, _) = variableDeclaration
        in try ignore(findVariable scope variableName);
raise(redeclaredVariableException(variableName))
           with Not_found ->
                        scope.variables <- variableDeclaration :: scope.variables

let getFunctionDeclaration scope functionName = List.find (function (name, _, _) -> name =
functionName) scope.functionsList

let isFunctionInEnv scope functionName =
```

```
        try (ignore (List.find (function (name, _, _) -> name = functionName)
scope.functionsList)); true
                with Not_found -> false

let updateGlobalScopeWithFunction scope functionName returnType parameterList =
        try ignore (List.find (function (name, _, _) -> name = functionName) scope.functionsList)
                with Not_found ->
                        scope.functionsList <- (functionName, returnType, parameterList) ::
scope.functionsList

let emptyEnviroment =
        let emptyScope = { variables = [] }
        in { scope = emptyScope }

let emptyGlobalEnvironment =
        let emptyGlobalScope = { functionsList = [] }
        in { globalScope = emptyGlobalScope }

let convertParamToVarDec = function
        Parameter(variableType, variableName) -> (variableName, variableType)

let getExpressionType = function
        | TypedBoolean(_, expressionType) -> OtherType(expressionType)
        | TypedNumber(_, expressionType) -> OtherType(expressionType)
        | TypedIdentifier(_, expressionType) -> OtherType(expressionType)
        | TypedUnop(_, expressionType, _) -> OtherType(expressionType)
        | TypedBinop(_, expressionType, _, _) -> OtherType(expressionType)
        | TypedSet(expressionType, _) -> OtherType(expressionType)
        | TypedFunctionCall(expressionType, _, _) -> expressionType
        | TypedVector(expressionType, _) -> OtherType(expressionType)
        | TypedVectorRange(_, _, _, _) -> OtherType(VectorLiteralType(NumberType("field",
"Integers")))
        | TypedCases(CasesLiteralType(expressionType), _, _) -> OtherType(expressionType)
        | TypedCases(_, _, _) -> raise(PlatoError("Wrong type for typed cases literal"))
        (*
        | TypedQuantifier(expressionType, _) -> expressionType
        *)

let canApplyAt = function
        | [NumberType("field", "Integers"); VectorLiteralType(NumberType("field", "Integers"))] ->
true
        | [VectorLiteralType(NumberType("field", "Integers")); NumberType("field", "Integers")] ->
true
```

```
        | [VectorLiteralType(NumberType("field", "Integers"));
VectorLiteralType(NumberType("field", "Integers"))] -> true
        | _ -> false

let canApplyNot = function
        | [BooleanType] -> true
        | [VectorLiteralType(BooleanType)] -> true
        | _ -> false

let canApplyOr = function
        | [BooleanType; BooleanType] -> true
        | [VectorLiteralType(BooleanType); BooleanType] -> true
        | [BooleanType; VectorLiteralType(BooleanType)] -> true
        | _ -> false

let canApplyAnd = function
        | [BooleanType; BooleanType] -> true
        | [VectorLiteralType(BooleanType); BooleanType] -> true
        | [BooleanType; VectorLiteralType(BooleanType)] -> true
        | _ -> false

let canApplyNegation = function
        | [NumberType(_, _)] -> true
        | [VectorLiteralType(NumberType(_, _))] -> true
        | _ -> false

let canApplyPlus = function
        | [NumberType(_, numberType1); NumberType(_, numberType2)] -> (numberType1 =
numberType2)
        | [SetLiteralType(arg1); SetLiteralType(arg2)] -> (arg1=arg2)
        | [VectorLiteralType(NumberType(_, numberType1)); NumberType(_, numberType2)] ->
(numberType1 = numberType2)
        | [NumberType(_, numberType1); VectorLiteralType(NumberType(_, numberType2))] ->
(numberType1 = numberType2)
        | _ -> false

let canApplyMinus = function
        | [NumberType(_, numberType1); NumberType(_, numberType2)] -> (numberType1 =
numberType2)
        | [VectorLiteralType(NumberType(_, numberType1)); NumberType(_, numberType2)] ->
(numberType1 = numberType2)
        | [NumberType(_, numberType1); VectorLiteralType(NumberType(_, numberType2))] ->
(numberType1 = numberType2)
```

```
        | _ -> false

let canApplyTimes = function
        | [NumberType(extendedGroupType1, numberType1);
NumberType(extendedGroupType2, numberType2)] ->
                (numberType1 = numberType2) && ((extendedGroupType1 = "ring") ||
(extendedGroupType1 = "field")) && ((extendedGroupType2 = "ring") || (extendedGroupType2 =
"field"))
 | [VectorLiteralType(NumberType(extendedGroupType1, numberType1));
NumberType(extendedGroupType2, numberType2)] ->
                (numberType1 = numberType2) && ((extendedGroupType1 = "ring") ||
(extendedGroupType1 = "field")) && ((extendedGroupType2 = "ring") || (extendedGroupType2 =
"field"))
        | [NumberType(extendedGroupType1, numberType1);
VectorLiteralType(NumberType(extendedGroupType2, numberType2))] ->
                (numberType1 = numberType2) && ((extendedGroupType1 = "ring") ||
(extendedGroupType1 = "field")) && ((extendedGroupType2 = "ring") || (extendedGroupType2 =
"field"))
        | [SetLiteralType(arg1); SetLiteralType(arg2)] -> arg1=arg2
        | _ -> false

let canApplyDivide = function
        | [NumberType(extendedGroupType1, numberType1);
NumberType(extendedGroupType2, numberType2)] ->
                (numberType1 = numberType2) && (extendedGroupType1 = "field") &&
(extendedGroupType2 = "field")
  | [VectorLiteralType(NumberType(extendedGroupType1, numberType1));
NumberType(extendedGroupType2, numberType2)] ->
                                (numberType1 = numberType2) && (extendedGroupType1 =
"field") && (extendedGroupType2 = "field")
        | [NumberType(extendedGroupType1, numberType1);
VectorLiteralType(NumberType(extendedGroupType2, numberType2))] ->
                                                (numberType1 = numberType2) &&
(extendedGroupType1 = "field") && (extendedGroupType2 = "field")
        | _ -> false

let canApplyMod = function
        | [NumberType(_, "Integers"); NumberType(_, "Integers")] -> true
        | [VectorLiteralType(NumberType(_, "Integers")); NumberType(_, "Integers")] -> true
        | [NumberType(_, "Integers"); VectorLiteralType(NumberType(_, "Integers"))] -> true
        | _ -> false

let canApplyRaise = function
```

```
        | [NumberType(_, "Integers"); NumberType(_, "Integers")] -> true
        | [SetLiteralType(arg1); SetLiteralType(arg2)] -> arg1=arg2
        | [VectorLiteralType(NumberType(_, "Integers")); NumberType(_, "Integers")] -> true
        | [NumberType(_, "Integers"); VectorLiteralType(NumberType(_, "Integers"))] -> true
        | _ -> false

let canApplyLessThan = function
        | [NumberType(_, "Integers"); NumberType(_, "Integers")] -> true
        | [VectorLiteralType(NumberType(_, "Integers")); NumberType(_, "Integers")] -> true
        | [NumberType(_, "Integers"); VectorLiteralType(NumberType(_, "Integers"))] -> true
        | _ -> false

let canApplyLessThanOrEqual = function
        | [NumberType(_, "Integers"); NumberType(_, "Integers")] -> true
        | [VectorLiteralType(NumberType(_, "Integers")); NumberType(_, "Integers")] -> true
        | [NumberType(_, "Integers"); VectorLiteralType(NumberType(_, "Integers"))] -> true
        | _ -> false

let canApplyGreaterThan = function
        | [NumberType(_, "Integers"); NumberType(_, "Integers")] -> true
        | [VectorLiteralType(NumberType(_, "Integers")); NumberType(_, "Integers")] -> true
        | [NumberType(_, "Integers"); VectorLiteralType(NumberType(_, "Integers"))] -> true
        | _ -> false

let canApplyGreaterThanOrEqual = function
        | [NumberType(_, "Integers"); NumberType(_, "Integers")] -> true
        | [VectorLiteralType(NumberType(_, "Integers")); NumberType(_, "Integers")] -> true
        | [NumberType(_, "Integers"); VectorLiteralType(NumberType(_, "Integers"))] -> true
        | _ -> false

let canApplyEqual = function
        | [NumberType(_, "Integers"); NumberType(_, "Integers")] -> true
        | [VectorLiteralType(NumberType(_, "Integers")); NumberType(_, "Integers")] -> true
        | [NumberType(_, "Integers"); VectorLiteralType(NumberType(_, "Integers"))] -> true
        | _ -> false

let canApplySetDifference = function
        | [SetLiteralType(arg1); SetLiteralType(arg2)] -> arg1=arg2
        | _ -> false

let canApplyVectorAccess = function
        | [VectorLiteralType(_); arg2] ->
```

```
                (canCast arg2 (NumberType("field", "Integers"))) || (canCast arg2
(VectorLiteralType(BooleanType)))
        | _ -> false

let canApplyOperator inputTypeList operatorType =
        try (ignore (List.find (fun p -> p=VoidType) inputTypeList)); false
                with Not_found ->
                        let pltInputTypeList = List.map (fun elem -> match elem with

OtherType(pltType) -> pltType

| _ -> raise(DebugException("If this point was reached, cannot have voidType"))) inputTypeList
                                        in (match operatorType with
                                            | At -> canApplyAt pltInputTypeList
                                                                | Not -> canApplyNot pltInputTypeList
                                                                | And -> canApplyAnd pltInputTypeList
                                                                | Or -> canApplyOr pltInputTypeList
                                                                | Negation -> canApplyNegation
pltInputTypeList
                                                        | Plus -> canApplyPlus pltInputTypeList
                                                                | Minus -> canApplyMinus pltInputTypeList
                                                                | Times -> canApplyTimes pltInputTypeList
                                                                | Divide -> canApplyDivide pltInputTypeList
                                                                | Mod -> canApplyMod pltInputTypeList
                                                                | Raise -> canApplyRaise pltInputTypeList
                                                                | LessThan -> canApplyLessThan
pltInputTypeList
                                                                | LessThanOrEqual ->
canApplyLessThanOrEqual pltInputTypeList
                                                                | GreaterThan -> canApplyGreaterThan
pltInputTypeList
                                                                | GreaterThanOrEqual ->
canApplyGreaterThanOrEqual pltInputTypeList
                                                                | Equal -> canApplyEqual pltInputTypeList
                                                                | SetDifference -> canApplySetDifference
pltInputTypeList
                                                                | VectorAccess -> canApplyVectorAccess
pltInputTypeList)

let getOperatorReturnType inputTypes operatorType =
        let pltInputTypeList = List.map (fun elem -> match elem with

OtherType(pltType) -> pltType
```

```
| _ -> raise(DebugException("If this point was reached, cannot have voidType"))) inputTypes
        in (match operatorType with
                | At ->
                        (match pltInputTypeList with
                         | [VectorLiteralType(NumberType("field", "Integers")); NumberType("field",
"Integers")] -> VectorLiteralType(NumberType("field", "Integers"))
                         | [NumberType("field", "Integers"); VectorLiteralType(NumberType("field",
"Integers"))] -> VectorLiteralType(NumberType("field", "Integers"))
                | [VectorLiteralType(NumberType("field", "Integers"));
VectorLiteralType(NumberType("field", "Integers"))] -> VectorLiteralType(NumberType("field",
"Integers"))
                         | _ -> raise(PlatoError("Less than must have exactly have two input
types")))
                | Not ->
                        (match pltInputTypeList with
                         | [BooleanType] -> BooleanType
                         | [VectorLiteralType(BooleanType)] -> VectorLiteralType(BooleanType)
                         | _ -> raise(PlatoError("Not must have exactly have one boolean input
type")))
                | And ->
                        (match pltInputTypeList with
                         | [BooleanType; BooleanType] -> BooleanType
                         | [VectorLiteralType(BooleanType); BooleanType] ->
VectorLiteralType(BooleanType)
                         | [BooleanType; VectorLiteralType(BooleanType)] ->
VectorLiteralType(BooleanType)
                         | _ -> raise(PlatoError("And must have exactly have two boolean input
types")))
                | Or ->
                        (match pltInputTypeList with
                         | [BooleanType; BooleanType] -> BooleanType
                         | [VectorLiteralType(BooleanType); BooleanType] ->
VectorLiteralType(BooleanType)
                         | [BooleanType; VectorLiteralType(BooleanType)] ->
VectorLiteralType(BooleanType)
                         | [VectorLiteralType(BooleanType); VectorLiteralType(BooleanType)] ->
VectorLiteralType(BooleanType)
                         | _ -> raise(PlatoError("Or must have exactly have two boolean input
types")))
                | Negation ->
                        (match pltInputTypeList with
                         | [inputType] -> inputType
```

```
                        | _ -> raise(PlatoError("Negation must have exactly have one input
type")))
                | Plus ->
                  (match pltInputTypeList with
                        | [VectorLiteralType(inputType1); inputType2] ->
VectorLiteralType(inputType1)
                        | [inputType1; VectorLiteralType(inputType2)] ->
VectorLiteralType(inputType2)
                        | [inputType1; inputTyp2] -> inputType1
                        | _ -> raise(PlatoError("Plus must have exactly have two input types")))
                | Minus ->
                  (match pltInputTypeList with
                        | [VectorLiteralType(inputType1); inputType2] ->
VectorLiteralType(inputType1)
                        | [inputType1; VectorLiteralType(inputType2)] ->
VectorLiteralType(inputType2)
                        | [inputType1; inputTyp2] -> inputType1
                        | _ -> raise(PlatoError("Minus must have exactly have two input types")))
                | Times ->
                  (match pltInputTypeList with
                        | [VectorLiteralType(inputType1); inputType2] ->
VectorLiteralType(inputType1)
                        | [inputType1; VectorLiteralType(inputType2)] ->
VectorLiteralType(inputType2)
                        | [inputType1; inputTyp2] -> inputType1
                        | _ -> raise(PlatoError("Times must have exactly have two input types")))
                | Divide ->
                (match pltInputTypeList with
                        | [VectorLiteralType(inputType1); inputType2] ->
VectorLiteralType(inputType1)
                        | [inputType1; VectorLiteralType(inputType2)] ->
VectorLiteralType(inputType2)
                        | [inputType1; inputTyp2] -> inputType1
                        | _ -> raise(PlatoError("Divide must have exactly have two input types")))
                | Mod ->
              (match pltInputTypeList with
                        | [VectorLiteralType(inputType1); inputType2] ->
VectorLiteralType(inputType1)
                        | [inputType1; VectorLiteralType(inputType2)] ->
VectorLiteralType(inputType2)
                        | [inputType1; inputTyp2] -> inputType1
                        | _ -> raise(PlatoError("Mod must have exactly have two input types")))
                | Raise ->
```

```
                        (match pltInputTypeList with
                         | [VectorLiteralType(inputType1); inputType2] ->
VectorLiteralType(inputType1)
                         | [inputType1; VectorLiteralType(inputType2)] ->
VectorLiteralType(inputType2)
                         | [inputType1; inputTyp2] -> inputType1
                         | _ -> raise(PlatoError("Raise must have exactly have two input types")))
                | LessThan ->
                        (match pltInputTypeList with
                         | [VectorLiteralType(NumberType("field", "Integers")); NumberType("field",
"Integers")] -> VectorLiteralType(BooleanType)
                         | [NumberType("field", "Integers"); NumberType("field", "Integers")] ->
BooleanType
                         | _ -> raise(PlatoError("Less than must have exactly have two input
types")))
                | LessThanOrEqual ->
                  (match pltInputTypeList with
                         | [VectorLiteralType(NumberType("field", "Integers")); NumberType("field",
"Integers")] -> VectorLiteralType(BooleanType)
                         | [NumberType("field", "Integers"); NumberType("field", "Integers")] ->
BooleanType
                         | _ -> raise(PlatoError("Less than must have exactly have two input
types")))
                | GreaterThan ->
                        (match pltInputTypeList with
                         | [VectorLiteralType(NumberType("field", "Integers")); NumberType("field",
"Integers")] -> VectorLiteralType(BooleanType)
                         | [NumberType("field", "Integers"); NumberType("field", "Integers")] ->
BooleanType
                         | _ -> raise(PlatoError("Less than must have exactly have two input
types")))
                | GreaterThanOrEqual ->
                        (match pltInputTypeList with
                         | [VectorLiteralType(NumberType("field", "Integers")); NumberType("field",
"Integers")] -> VectorLiteralType(BooleanType)
                         | [NumberType("field", "Integers"); NumberType("field", "Integers")] ->
BooleanType
                         | _ -> raise(PlatoError("Less than must have exactly have two input
types")))
                | Equal ->
                        (match pltInputTypeList with
                         | [VectorLiteralType(NumberType("field", "Integers")); NumberType("field",
"Integers")] -> VectorLiteralType(BooleanType)
```

```
                        | [NumberType("field", "Integers"); NumberType("field", "Integers")] ->
BooleanType
                        | _ -> raise(PlatoError("Less than must have exactly have two input
types")))
                | SetDifference ->
                        (match pltInputTypeList with
                        | [inputType1; inputType2] -> inputType1
                        | _ -> raise(PlatoError("Set difference must have exactly have two input
types")))
                | VectorAccess ->
                        (match pltInputTypeList with
                        | [VectorLiteralType(arg1); arg2] ->
                                (if (canCast arg2 (NumberType("field", "Integers")))
                                 then arg1
                                 else VectorLiteralType(arg1))
                        | _ -> raise(PlatoError("VectorAccess must have exactly have two input
types, and the first type must be a VectorLiteral")))

                )

let getOperatorCallClass inputTypeList = function
        | At -> "VectorLiterals"
        | Not -> "Booleans"
        | And -> "Booleans"
        | Or -> "Booleans"
        | Negation ->
                (match inputTypeList with
                 | [NumberType(_, groupName)] -> groupName
                 | [VectorLiteralType(NumberType(_, groupName))] -> groupName
                 | _ -> raise(operatorException Negation inputTypeList))
  | Plus ->
                (match inputTypeList with
                 | [VectorLiteralType(NumberType(_, groupName)); NumberType(_, _)] ->
groupName
                 | [NumberType(_, _); VectorLiteralType(NumberType(_, groupName))] ->
groupName
                 | [NumberType(_, groupName); _] -> groupName
                 | [SetLiteralType(_); _] -> "SetLiterals"
                 | _ -> raise(operatorException Plus inputTypeList))
        | Minus ->
                (match inputTypeList with
                 | [VectorLiteralType(NumberType(_, groupName)); NumberType(_, _)] ->
groupName
```

```
                | [NumberType(_, _); VectorLiteralType(NumberType(_, groupName))] ->
groupName
                | [NumberType(_, groupName); _] -> groupName
                | _ -> raise(operatorException Minus inputTypeList))
        | Times ->
                (match inputTypeList with
                | [VectorLiteralType(NumberType(_, groupName)); NumberType(_, _)] ->
groupName
                | [NumberType(_, _); VectorLiteralType(NumberType(_, groupName))] ->
groupName
                | [NumberType(_, groupName); _] -> groupName
                | [SetLiteralType(_); _] -> "SetLiterals"
                | _ -> raise(operatorException Times inputTypeList))
        | Divide ->
                (match inputTypeList with
                | [VectorLiteralType(NumberType(_, groupName)); NumberType(_, _)] ->
groupName
                | [NumberType(_, _); VectorLiteralType(NumberType(_, groupName))] ->
groupName
                | [NumberType(_, groupName); _] -> groupName
                | _ -> raise(operatorException Divide inputTypeList))
        | Mod ->
                (match inputTypeList with
                | [VectorLiteralType(NumberType(_, groupName)); NumberType(_, _)] ->
groupName
                | [NumberType(_, _); VectorLiteralType(NumberType(_, groupName))] ->
groupName
                | [NumberType(_, groupName); _] -> groupName
                | _ -> raise(operatorException Mod inputTypeList))
        | Raise ->
                (match inputTypeList with
                | [VectorLiteralType(NumberType(_, groupName)); NumberType(_, _)] ->
groupName
                | [NumberType(_, _); VectorLiteralType(NumberType(_, groupName))] ->
groupName
                | [NumberType(_, groupName); _] -> groupName
                | [SetLiteralType(_); _] -> "SetLiterals"
                | _ -> raise(operatorException Raise inputTypeList))
        | LessThan ->
                (match inputTypeList with
                | [VectorLiteralType(NumberType(_, groupName)); NumberType(_, _)] ->
groupName
```

```
                | [NumberType(_, _); VectorLiteralType(NumberType(_, groupName))] ->
groupName
                | [NumberType(_, groupName); _] -> groupName
                | _ -> raise(operatorException LessThan inputTypeList))
        | LessThanOrEqual ->
                (match inputTypeList with
                | [VectorLiteralType(NumberType(_, groupName)); NumberType(_, _)] ->
groupName
                | [NumberType(_, _); VectorLiteralType(NumberType(_, groupName))] ->
groupName
                | [NumberType(_, groupName); _] -> groupName
                | _ -> raise(operatorException LessThanOrEqual inputTypeList))
        | GreaterThan ->
                (match inputTypeList with
                | [VectorLiteralType(NumberType(_, groupName)); NumberType(_, _)] ->
groupName
                | [NumberType(_, _); VectorLiteralType(NumberType(_, groupName))] ->
groupName
                | [NumberType(_, groupName); _] -> groupName
                | _ -> raise(operatorException GreaterThan inputTypeList))
        | GreaterThanOrEqual ->
                (match inputTypeList with
                | [VectorLiteralType(NumberType(_, groupName)); NumberType(_, _)] ->
groupName
                | [NumberType(_, _); VectorLiteralType(NumberType(_, groupName))] ->
groupName
                | [NumberType(_, groupName); _] -> groupName
                | _ -> raise(operatorException GreaterThanOrEqual inputTypeList))
        | Equal ->
                (match inputTypeList with
                | [VectorLiteralType(NumberType(_, groupName)); NumberType(_, _)] ->
groupName
                | [NumberType(_, _); VectorLiteralType(NumberType(_, groupName))] ->
groupName
                | [NumberType(_, groupName); _] -> groupName
                | _ -> raise(operatorException Equal inputTypeList))
        | SetDifference ->
                (match inputTypeList with
                | [SetLiteralType(_); _] -> "SetLiterals"
                | _ -> raise(operatorException SetDifference inputTypeList))
        | VectorAccess ->
                (match inputTypeList with
                | [VectorLiteralType(_); _] -> ("VectorLiterals")
```

```
                | _ -> raise(operatorException VectorAccess inputTypeList))

let rec checkExpression globalEnv environment = function
        | Boolean(booleanValue) -> TypedBoolean(booleanValue, BooleanType)
        | Number(numberValue) -> TypedNumber(numberValue, NumberType("field", "Integers"))
        | Identifier(variableName) ->
                let variableDeclaration =
                                try findVariable environment.scope variableName
                        with Not_found -> raise (undeclaredVariableException variableName)
                in  let (_, variableType) = variableDeclaration
                        in TypedIdentifier(variableName, variableType)
        | Unop(unaryOperator, unopExpression) ->
                (let unaryExpression =  checkExpression globalEnv environment unopExpression
                in let expressionTypeList = [getExpressionType unaryExpression]
                  in if canApplyOperator expressionTypeList unaryOperator
                        then TypedUnop(unaryOperator, getOperatorReturnType
expressionTypeList unaryOperator, unaryExpression)
                        else
                            let pltExpressionTypeList = List.map (fun elem -> match elem with

OtherType(pltType) -> pltType

| _ -> raise(DebugException("If this point was reached, cannot have voidType")))
expressionTypeList
                            in raise(operatorException unaryOperator pltExpressionTypeList))
        | Binop(binaryOperator, binaryExpression1, binaryExpression2) ->
                (let binaryExpression1 = checkExpression globalEnv environment
binaryExpression1
                and binaryExpression2 = checkExpression globalEnv environment
binaryExpression2
        in let expressionTypeList = [getExpressionType binaryExpression1; getExpressionType
binaryExpression2]
                  in if canApplyOperator expressionTypeList binaryOperator
                        then TypedBinop(binaryOperator, getOperatorReturnType
expressionTypeList binaryOperator, binaryExpression1, binaryExpression2)
                        else
                            let pltExpressionTypeList = List.map (fun elem -> match elem with

OtherType(pltType) -> pltType

| _ -> raise(DebugException("If this point was reached, cannot have voidType")))
expressionTypeList
```

```
                                    in raise(operatorException binaryOperator
pltExpressionTypeList))
        | SetLiteral(setopExpressionList) ->
                (match setopExpressionList with
                        [] -> TypedSet(SetLiteralType(NeutralType), [])
                        | _ -> (let setExpressionList =  List.map (checkExpression globalEnv
environment) setopExpressionList
                                in let expressionTypeList = List.map getExpressionType
setExpressionList
                                        in let headExpressionType = List.hd expressionTypeList
                                                in if allTrue (List.map (fun arg1 ->
(headExpressionType=arg1)) expressionTypeList)
                                                                then
TypedSet(SetLiteralType(extractPltTypeFromFuncType (List.hd expressionTypeList)),
setExpressionList)
                                                                else
raise(heterogeneousSetLiteralException (List.map extractPltTypeFromFuncType
expressionTypeList) )))
        | VectorLiteral(vectoropExpressionList) ->
                (match vectoropExpressionList with
                        [] -> TypedVector(VectorLiteralType(NeutralType), [])
                        | _ -> (let vectorExpressionList =  List.map (checkExpression globalEnv
environment) vectoropExpressionList
                                in let expressionTypeList = List.map getExpressionType
vectorExpressionList
                                        in let headExpressionType = List.hd expressionTypeList
                                                in if allTrue (List.map (fun arg1 ->
(headExpressionType=arg1)) expressionTypeList)
                                                                then
TypedVector(VectorLiteralType(extractPltTypeFromFuncType (List.hd expressionTypeList)),
vectorExpressionList)
                                                                else
raise(heterogeneousVectorLiteralException (List.map extractPltTypeFromFuncType
expressionTypeList))))
        | CasesLiteral(casePairsList, defaultReturn) ->
                let defaultReturnExpression = checkExpression globalEnv environment
defaultReturn
                in let defaultReturnType = extractPltTypeFromFuncType (getExpressionType
defaultReturnExpression)
                in
                (match casePairsList with
                        [] -> TypedCases(CasesLiteralType(defaultReturnType), [],
defaultReturnExpression)
```

```
                        | _ -> (let caseExpressionList = List.map
                                (fun (x, y) ->
                                                        (checkExpression globalEnv environment x,
                                                        checkExpression globalEnv environment y))
casePairsList
                                        in let expressionTypeList = List.map
                                                        (fun (x, y) ->
                                                                (extractPltTypeFromFuncType
(getExpressionType x),

extractPltTypeFromFuncType (getExpressionType y)))
                                                        caseExpressionList
                                        in ignore (List.map (fun (x,y)-> if x<>defaultReturnType ||
y<>BooleanType then raise(PlatoError("Cases literal has inconsistent types")))
expressionTypeList);
                                        TypedCases(CasesLiteralType(defaultReturnType),
caseExpressionList, defaultReturnExpression)
                ))


        | VectorRange(fromExpression, toExpression, byExpression) ->
                let fromExpression, toExpression, byExpression = (checkExpression globalEnv
environment fromExpression), (checkExpression globalEnv environment toExpression),
(checkExpression globalEnv environment byExpression)
          in let fromExpressionType, toExpressionType, byExpressionType =
(getExpressionType fromExpression), (getExpressionType toExpression), (getExpressionType
byExpression)
      in if canCast (extractPltTypeFromFuncType fromExpressionType) (NumberType("field",
"Integers"))
                                then if canCast (extractPltTypeFromFuncType
toExpressionType) (NumberType("field", "Integers"))
                                                then if canCast (extractPltTypeFromFuncType
byExpressionType) (NumberType("field", "Integers"))
                                                                then
TypedVectorRange(VectorLiteralType(NumberType("field", "Integers")), fromExpression,
toExpression, byExpression)
                                                                else
raise(castException (extractPltTypeFromFuncType byExpressionType) (NumberType("field",
"Integers")))
                                                else raise(castException
(extractPltTypeFromFuncType byExpressionType) (NumberType("field", "Integers")))
                                else raise(castException (extractPltTypeFromFuncType
byExpressionType) (NumberType("field", "Integers")))
```

```
	| FunctionCall(functionName, expressionList) ->
		if isFunctionInEnv globalEnv.globalScope functionName
		then
			let (_, functionType, parameterList) = getFunctionDeclaration
globalEnv.globalScope functionName
			in let listOfCheckedExpressions = List.map (checkExpression globalEnv
environment) expressionList
				in let listOfExpressionTypes = List.map
extractPltTypeFromFuncType (List.map getExpressionType listOfCheckedExpressions)
					in let listOfTypesOfParams = List.map (fun elem -> match
elem with

					Parameter(platoType, _) -> platoType) parameterList
						in if listOfExpressionTypes=listOfTypesOfParams
							then TypedFunctionCall(functionType,
functionName, listOfCheckedExpressions)
							else
raise(functionCallParameterTypeMismatchException functionName)
		else raise(functionNotFoundException functionName)

let getTypedStmtBlockReturnType = function
	TypedStatementBlock(typedStatementList) ->
		if (List.length typedStatementList)=0
		then VoidType
		else
		(* everything but last stmt should have VoidType, otherwise exception *)
			((ignore (List.map (fun stmt -> match stmt with
				TypedReturn(returnType, _) ->
raise(unreachableCodeException)
				| TypedIf(returnType, _, _, _, _) -> (match returnType with

			VoidType -> ()

			| _ -> raise(unreachableCodeException))
				| _ -> ()) (List.tl (List.rev typedStatementList))));
			let lastStmt = List.hd (List.rev typedStatementList)
			in  ( match lastStmt with
				TypedReturn(returnType, _) -> returnType
				| TypedIf(returnType, _, _, _, _) -> returnType
				| _ -> VoidType ))
let rec checkStatement globalEnv environment = function
	| VoidCall(expression) ->
		let typedExpression = checkExpression globalEnv environment expression
```

```
                in if (getExpressionType typedExpression) = VoidType
                  then TypedVoidCall(checkExpression globalEnv environment expression)
                       else raise(PlatoError("A bare statement can only contain a void function
call"))
        | Print(expression) -> TypedPrint(checkExpression globalEnv environment expression)
        | Return(expression) ->
                let checkedExpression = checkExpression globalEnv environment expression
                in TypedReturn(getExpressionType checkedExpression, checkedExpression)
        | If (expression, statementBlock, elseIfBlockList, elseBlock) ->
                let typedExpression = checkExpression globalEnv environment expression
                in let expressionType = getExpressionType typedExpression
                        in if canCast (extractPltTypeFromFuncType expressionType)
BooleanType
                                then
                                        let checkedStatementBlock = checkStatementBlock
globalEnv environment statementBlock
                                        in let typeOfLastStmtInBlock =
getTypedStmtBlockReturnType checkedStatementBlock
                                                in let checkedElseIfs = List.map (checkElseIfBlock
globalEnv environment) elseIfBlockList
                                                        in let checkedElse = checkElseBlock
globalEnv environment elseBlock
                                                                in if typeOfLastStmtInBlock =
VoidType
                                                                        then TypedIf(VoidType,
typedExpression, checkedStatementBlock, checkedElseIfs, checkedElse)
                                                                else
                                                                        let
checkedElseTypedStmtBlock = (match checkedElse with

                                        TypedElseBlock(typedStatementBlock) ->
typedStatementBlock)
                                                                                in let
checkedElseIfsTypedStmtBlocks = List.map (fun element -> match element with

                                                                TypedElseIfBlock(_,
typedStatementBlock) -> typedStatementBlock

                                                                ) checkedElseIfs
                                                                        in let
typesOfLastStmtsInElseIfAndElseBlocks = (getTypedStmtBlockReturnType
checkedElseTypedStmtBlock)::(List.map getTypedStmtBlockReturnType
checkedElseIfsTypedStmtBlocks)
```

```
                                                                    in if
List.fold_left (fun booleanValue expressionType -> booleanValue && (expressionType =
typeOfLastStmtInBlock)) true typesOfLastStmtsInElseIfAndElseBlocks
                                                                    then
TypedIf(typeOfLastStmtInBlock, typedExpression, checkedStatementBlock, checkedElseIfs,
checkedElse)
                                                                    else
TypedIf(VoidType, typedExpression, checkedStatementBlock, checkedElseIfs, checkedElse)
                                    else raise(noneBooleanExpressionInIf
(extractPltTypeFromFuncType expressionType))
        | IfNoElse (expression, statementBlock, elseIfBlockList) ->
                let typedExpression = checkExpression globalEnv environment expression
                in let expressionType = getExpressionType typedExpression
                        in if canCast (extractPltTypeFromFuncType expressionType)
BooleanType
                                    then TypedIfNoElse(typedExpression, checkStatementBlock
globalEnv environment statementBlock, List.map (checkElseIfBlock globalEnv environment)
elseIfBlockList)
                                    else raise(noneBooleanExpressionInIf
(extractPltTypeFromFuncType expressionType))
  | Assignment(variableName, newValue) ->
                let variableIdentifier = Identifier(variableName)
                in let variableDetails = checkExpression globalEnv environment variableIdentifier
                        in let expressionDetails = checkExpression globalEnv environment
newValue
                                in let expressionType, variableType = (getExpressionType
expressionDetails), (getExpressionType variableDetails)
                                                        in if canCast (extractPltTypeFromFuncType
expressionType) (extractPltTypeFromFuncType variableType)
                                                    then TypedAssignment((variableName,
(extractPltTypeFromFuncType variableType)), expressionDetails)
                                                        else raise(castException
(extractPltTypeFromFuncType expressionType) (extractPltTypeFromFuncType variableType))
        | VectorAssignment(variableName, variableIndex, newValue) ->
                let variableIdentifier = Identifier(variableName)
                 in let variableDetails, variableIndexDetails, expressionDetails = (checkExpression
globalEnv environment variableIdentifier), (checkExpression globalEnv environment
variableIndex), (checkExpression globalEnv environment newValue)
                    in let variableIndexType, expressionType, variableType = (getExpressionType
variableIndexDetails), (getExpressionType expressionDetails), (getExpressionType
variableDetails)
                                    in (match (extractPltTypeFromFuncType variableType) with
```

```
                                                | VectorLiteralType(variableSubType) -> if ((canCast
(extractPltTypeFromFuncType expressionType) variableSubType)
                                                                && (canCast
(extractPltTypeFromFuncType variableIndexType) (NumberType("field", "Integers"))))
                                            then TypedVectorAssignment((variableName,
(extractPltTypeFromFuncType variableType)), variableIndexDetails, expressionDetails)
                                                        else raise(castException
(extractPltTypeFromFuncType expressionType) (extractPltTypeFromFuncType variableType))
                                                                | _ -> raise(PlatoError("Cannot use
vector assignment for non-vector.")))
        | Declaration(variableType, variableName, newValue) ->
                let expressionDetails = checkExpression globalEnv environment newValue
                  in let expressionType = (getExpressionType expressionDetails)
                          in if canCast (extractPltTypeFromFuncType expressionType)
variableType
                             then (updateScope environment.scope (variableName, variableType);
                                                TypedDeclaration((variableName,
variableType), expressionDetails))
                                          else raise(castException (extractPltTypeFromFuncType
expressionType) variableType)
and checkStatementBlock globalEnv environment = function
          StatementBlock(statementList) -> TypedStatementBlock(List.map (checkStatement
globalEnv environment) statementList)
and checkElseIfBlock globalEnv environment = function
        ElseIfBlock(expression, statementBlock) ->
                let typedExpression = checkExpression globalEnv environment expression
                in let expressionType = getExpressionType typedExpression
                        in if canCast (extractPltTypeFromFuncType expressionType)
BooleanType
                                then TypedElseIfBlock(typedExpression, checkStatementBlock
globalEnv environment statementBlock)
                                else raise(noneBooleanExpressionInElseIf
(extractPltTypeFromFuncType expressionType))
and checkElseBlock globalEnv environment = function
        ElseBlock(statementBlock) ->
                TypedElseBlock(checkStatementBlock globalEnv environment statementBlock)

let checkMainBlock globalEnv = function
          MainBlock(mainBlock) -> TypedMainBlock(checkStatementBlock globalEnv
emptyEnviroment mainBlock)

let rec getReturnStmtsHelper = function
        [] -> []
```

```
        | Return(expression)::[] -> [Return(expression)]
        | Return(expression)::tail -> Return(expression)::getReturnStmtsHelper tail
        | _::tail -> getReturnStmtsHelper tail

let getReturnStmts = function
        StatementBlock(statementList) ->
                getReturnStmtsHelper(statementList)

let getLastStmtInBlock = function
        StatementBlock(statementList) -> List.hd (List.rev statementList)

let functionTypeToString = function
        VoidType -> "void"
        | OtherType(pltType) -> typeToString pltType
let checkFunctionBlock globalEnv = function
          FunctionDeclaration(functionHeader, statementBlock) ->
                let functionReturnType = functionHeader.returnType
                        in let functionEnvironment = emptyEnviroment
                          in (ignore (List.map (updateScope functionEnvironment.scope)
(List.map convertParamToVarDec functionHeader.parameters)));
                                        (ignore (updateGlobalScopeWithFunction
globalEnv.globalScope functionHeader.functionName functionHeader.returnType
functionHeader.parameters));
                                    let checkedStatementBlock = checkStatementBlock globalEnv
functionEnvironment statementBlock
                                        in let stmtBlockReturnType =
getTypedStmtBlockReturnType checkedStatementBlock
                                                in if canFunctionCast stmtBlockReturnType
functionReturnType
                                                        then

TypedFunctionDeclaration(functionHeader, checkedStatementBlock)
                                                        else raise(incompatibleTypesReturnStmt
functionHeader.functionName (functionTypeToString functionHeader.returnType)
(functionTypeToString stmtBlockReturnType))

let rec generateTableHelper rowElements columnElements tableFunction tableSoFar =
        match rowElements with
        | [] -> List.rev tableSoFar
        | head::tail ->
                let paritalTableFunction = (fun input2 -> evaluateSimpleBinaryFunction head
input2 tableFunction)
```

```
            in generateTableHelper tail columnElements tableFunction ((List.map
paritalTableFunction columnElements)::tableSoFar)

let generateTable tableElements tableFunction = generateTableHelper tableElements
tableElements tableFunction []

let rec getGroupIdentityHelper groupName groupElements groupTable currentIndex =
        match groupTable with
        | [] -> raise(identityException groupName)
        | head::tail ->
                if head = groupElements
                then List.nth groupElements currentIndex
                else getGroupIdentityHelper groupName groupElements tail (currentIndex + 1)

let getGroupIdentity groupName groupElements groupTable = getGroupIdentityHelper
groupName groupElements groupTable 0

let rec findInverseIndexHelper groupName element groupIdentity currentIndex = function
        | [] -> raise(inverseException groupName element)
        | head::tail ->
                if head = groupIdentity
                then currentIndex
                else findInverseIndexHelper groupName element groupIdentity (currentIndex + 1)
tail

let findInverseIndex groupName element groupIdentity additionResults = findInverseIndexHelper
groupName element groupIdentity 0 additionResults

let rec generateInverseListHelper groupName groupElements remainingElements groupIdentity
remainingTable listSoFar =
        match remainingElements with
        | [] -> List.rev listSoFar
        | head::tail ->
                let headInverse = List.nth groupElements (findInverseIndex groupName head
groupIdentity (List.hd remainingTable))
                in generateInverseListHelper groupName groupElements tail groupIdentity (List.tl
remainingTable) (headInverse :: listSoFar)

let generateInverseList groupName groupElements groupIdentity groupTable =
generateInverseListHelper groupName groupElements groupElements groupIdentity groupTable
[]

let print_table table =
```

```
        ignore (List.map (fun intList -> ignore (List.map print_int intList); print_string "\n") table)

let rec isElement list element =
        match list with
        | [] -> false
        | head::tail -> if element = head
                        then true
                                                                        else isElement tail element

let isClosed groupElements groupTable = allTrue (List.map (fun intList -> allTrue (List.map
(isElement groupElements) intList)) groupTable)

let rec getIndexHelper element list startIndex =
        match list with
        | [] -> raise Not_found
        | head::tail ->
                if element = head
          then startIndex
          else getIndexHelper element tail (startIndex + 1)

let getIndex element list = getIndexHelper element list 0

let checkAssociative a b c groupElements groupTable =
        let groupTimes = fun a b -> List.nth (List.nth groupTable (getIndex a groupElements))
(getIndex b groupElements)
        in let starResult = groupTimes (groupTimes a b) c
           in let circleResult = groupTimes a (groupTimes b c)
                   in starResult = circleResult

let rec checkAssociativeList aList b c groupElements groupTable =
        match aList with
        | [] -> true
        | head::tail -> if checkAssociative head b c groupElements groupTable
                        then checkAssociativeList tail b c groupElements groupTable
                                else false

let rec checkAssociativeListPair aList bList c groupElements groupTable =
        match bList with
        | [] -> true
        | head::tail -> if checkAssociativeList aList head c groupElements groupTable
                        then checkAssociativeListPair aList tail c groupElements groupTable
                                else false
```

```
let rec checkAssociativeListTriple aList bList cList groupElements groupTable =
        match cList with
        | [] -> true
        | head::tail -> if checkAssociativeListPair aList bList head groupElements groupTable
                    then checkAssociativeListTriple aList bList tail groupElements groupTable
                        else false

let isAssociative groupElements groupTable = checkAssociativeListTriple groupElements
groupElements groupElements groupElements groupTable


let rec removeNthHelper n list acc =
        if n = 0
        then (List.rev acc) @ (List.tl list)
        else removeNthHelper (n - 1) (List.tl list) ((List.hd list)::acc)

let removeNth n list = removeNthHelper n list []

let rec isCommutative table =
        match table with
        | [] -> true
        | _ -> if List.hd table = List.map List.hd table
             then isCommutative (List.tl (List.map List.tl table))
                                    else false

let checkDistributive a b c groupElements additionTable multiplicationTable =
  let groupPlus = fun a b -> List.nth (List.nth additionTable (getIndex a groupElements)) (getIndex
b groupElements)
        in let groupTimes = fun a b -> List.nth (List.nth multiplicationTable (getIndex a
groupElements)) (getIndex b groupElements)
           in let starResult = groupTimes a (groupPlus b c)
              in let circleResult = groupPlus (groupTimes a b) (groupTimes a c)
                      in starResult = circleResult

let rec checkDistributiveList aList b c groupElements additionTable multiplicationTable  =
        match aList with
        | [] -> true
        | head::tail -> if checkDistributive head b c groupElements additionTable
multiplicationTable
                    then checkDistributiveList tail b c groupElements additionTable
multiplicationTable
                        else false
```

```
let rec checkDistributiveListPair aList bList c groupElements additionTable multiplicationTable =
        match bList with
        | [] -> true
        | head::tail -> if checkDistributiveList aList head c groupElements additionTable
multiplicationTable
                        then checkDistributiveListPair aList tail c groupElements additionTable
multiplicationTable
                                else false

let rec checkDistributiveListTriple aList bList cList groupElements additionTable
multiplicationTable =
        match cList with
        | [] -> true
        | head::tail -> if checkDistributiveListPair aList bList head groupElements additionTable
multiplicationTable
                        then checkDistributiveListTriple aList bList tail groupElements additionTable
multiplicationTable
                                else false

let distributes groupElements additionTable multiplicationTable = checkDistributiveListTriple
groupElements groupElements groupElements groupElements additionTable multiplicationTable

let checkExtendedGroupBlock = function
          | GroupDeclaration(GroupHeader(groupName), GroupBody(groupElements,
groupAdditionFunction)) ->
                    let groupElementList = evaluateSimpleSet groupElements
                          in let additionTable = generateTable groupElementList
groupAdditionFunction
                                    in if (isClosed groupElementList additionTable) && (isAssociative
groupElementList additionTable)
                                       then let additiveInverseList = generateInverseList groupName
groupElementList (getGroupIdentity groupName groupElementList additionTable) additionTable
                                          in TypedGroupDeclaration(groupName, groupElementList,
additionTable, additiveInverseList)
                                else raise(PlatoError("Group addition must be closed and
associative"))
  | ExtendedGroupDeclaration(RingHeader(groupName),
ExtendedGroupBody(GroupBody(groupElements, groupAdditionFunction),
extendedGroupMultiplicationFunction)) ->
                    let groupElementList = evaluateSimpleSet groupElements
                    in let additionTable = generateTable groupElementList groupAdditionFunction
                          in if (isClosed groupElementList additionTable) && (isAssociative
groupElementList additionTable) && (isCommutative additionTable)
```

```
                then let additiveInverseList = generateInverseList groupName
groupElementList (getGroupIdentity groupName groupElementList additionTable) additionTable
                    in let multiplicationTable = generateTable groupElementList
extendedGroupMultiplicationFunction
                        in if (isClosed groupElementList
multiplicationTable) && (isAssociative groupElementList multiplicationTable) && (distributes
groupElementList additionTable multiplicationTable)
                            then TypedRingDeclaration(groupName,
groupElementList, additionTable, additiveInverseList, multiplicationTable)
                            else raise(PlatoError("Ring
multiplication must be closed and associative and distribute over addition"))
                    else raise(PlatoError("Ring addition must be closed and
associative"))
    | ExtendedGroupDeclaration(FieldHeader(groupName),
ExtendedGroupBody(GroupBody(groupElements, groupAdditionFunction),
extendedGroupMultiplicationFunction)) ->
                let groupElementList = evaluateSimpleSet groupElements
                in let additionTable = generateTable groupElementList
groupAdditionFunction
                    in if (isClosed groupElementList additionTable) && (isAssociative
groupElementList additionTable) && (isCommutative additionTable)
                then let additiveIdentity = getGroupIdentity groupName groupElementList
additionTable
                        in let additiveInverseList = generateInverseList
groupName groupElementList additiveIdentity additionTable
                    in let multiplicationTable = generateTable groupElementList
extendedGroupMultiplicationFunction
                        in if (isClosed groupElementList
multiplicationTable) && (isAssociative groupElementList multiplicationTable) && (isCommutative
multiplicationTable) && (distributes groupElementList additionTable multiplicationTable)
                            then let
additiveIdentityIndex = getIndex additiveIdentity groupElementList
                                in let
multiplicitiveInverseList = generateInverseList groupName (removeNth additiveIdentityIndex
groupElementList)  (getGroupIdentity groupName groupElementList multiplicationTable)
(List.map (removeNth additiveIdentityIndex) (removeNth additiveIdentityIndex multiplicationTable))
                                    in
TypedFieldDeclaration(groupName, groupElementList, additionTable, additiveInverseList,
multiplicationTable, multiplicitiveInverseList, additiveIdentity)
                            else
raise(PlatoError("Field multiplication must be closed, associative, commutative and distribute
over addition"))
```

```
                                    else raise(PlatoError("Field addition must be closed and
associative"))

let checkProgram globalEnv = function
        Program(mainBlock, functionBlockList, extendedGroupBlockList) ->
            let checkedFunctionBlockList = List.map (checkFunctionBlock globalEnv)
functionBlockList
                        in TypedProgram(checkMainBlock globalEnv mainBlock,
checkedFunctionBlockList, List.map checkExtendedGroupBlock extendedGroupBlockList)

(* Convert Sast to Java Ast *)
let createJavaType = function
        | BooleanType -> JavaBooleanType
        | NumberType(_, _) -> JavaIntType
        | SetLiteralType(_) -> JavaSetLiteralType
        | VectorLiteralType(_) -> JavaVectorLiteralType
        | CasesLiteralType(_) -> JavaCasesLiteralType
        | NeutralType -> JavaNeutralType

let rec createJavaExpression = function
        | TypedBoolean(booleanValue, _) ->
JavaConstant(JavaValue(JavaBoolean(booleanValue)))
        | TypedNumber(numberValue, _)-> JavaConstant(JavaValue(JavaInt(numberValue)))
        | TypedIdentifier(variableName, _) -> JavaVariable(variableName)
        | TypedUnop(unaryOperator, operatorType, unopExpression) ->
            JavaCall(getOperatorCallClass [extractPltTypeFromFuncType
(getExpressionType unopExpression)] unaryOperator, operatorToString unaryOperator,
[createJavaExpression unopExpression])
        | TypedBinop(binaryOperator, operatorType, binaryExpression1, binaryExpression2) ->
            JavaCall(getOperatorCallClass [extractPltTypeFromFuncType
(getExpressionType binaryExpression1); extractPltTypeFromFuncType (getExpressionType
binaryExpression2)] binaryOperator, operatorToString binaryOperator, [createJavaExpression
binaryExpression1; createJavaExpression binaryExpression2])
        | TypedSet(setType, setExpressionList) ->
            JavaCall("SetLiterals", "newPlatoSet", List.map createJavaExpression
setExpressionList)
        | TypedFunctionCall(platoFunctionType, functionName, expressionList) ->
            JavaCall("", functionName, List.map createJavaExpression expressionList)
        | TypedVector(vectorType, vectorExpressionList) ->
            JavaCall("VectorLiterals", "newPlatoVector", List.map createJavaExpression
vectorExpressionList)
        | TypedVectorRange(vectorType, fromExpression, toExpression, byExpression) ->
```

```
                JavaCall("VectorLiterals", "newPlatoVectorRange", [createJavaExpression
fromExpression; createJavaExpression toExpression; createJavaExpression byExpression])
        | TypedCases(casesType, casesExpressionsList, defaultExpression) ->
                JavaTernaryChain(List.map (fun (x,y) -> (createJavaExpression x,
createJavaExpression y)) casesExpressionsList, createJavaExpression defaultExpression)

let rec createJavaStatement = function
        | TypedVoidCall(expression) -> JavaStatement(JavaCall("", "", [createJavaExpression
expression]))
        | TypedPrint(expression) -> JavaStatement(JavaCall("System.out", "println",
[createJavaExpression expression]))
        | TypedReturn(_, expression) -> JavaStatement(JavaReturn(createJavaExpression
expression))
        | TypedIf(_, typedExpression, typedStatementBlock, typedElseIfBlockList,
typedElseBlock) ->
                                        JavaStatement(
                                                JavaIf(createJavaExpression typedExpression,
                                                                createJavaBlock
typedStatementBlock,
                                                                List.map createJavaElseIf
typedElseIfBlockList,
                                                                createJavaElse typedElseBlock))
        | TypedIfNoElse(typedExpression, typedStatementBlock, typedElseIfBlockList) ->
                                        JavaStatement(
                                                JavaIfNoElse(createJavaExpression
typedExpression,
                                                                createJavaBlock
typedStatementBlock,
                                                                List.map createJavaElseIf
typedElseIfBlockList
                                                        )
                                        )
        | TypedAssignment((variableName, variableType), newValue) ->
JavaStatement(JavaAssignment(variableName, createJavaExpression newValue))
        | TypedVectorAssignment((variableName, variableType), indexValue, newValue) ->
JavaStatement(JavaVectorAssignment(variableName, createJavaExpression indexValue,
createJavaExpression newValue))
        | TypedDeclaration((variableName, variableType), newValue) ->
JavaStatement(JavaDeclaration(createJavaType variableType, variableName,
Some(createJavaExpression newValue)))

and createJavaElseIf = function
```

```
        TypedElseIfBlock(typedExpression, typedStatementBlock) ->
JavaElseIf(createJavaExpression typedExpression, createJavaBlock typedStatementBlock)
and createJavaElse = function
        TypedElseBlock(typedStatementBlock) -> JavaElse(createJavaBlock
typedStatementBlock)
and createJavaBlock = function
        TypedStatementBlock(typedStatementList) -> JavaBlock(List.map createJavaStatement
typedStatementList)

let createJavaFunction = function
          TypedFunctionDeclaration(functionHeader, TypedStatementBlock(typedStatementList))
->
                        JavaFunction(functionHeader, JavaBlock(List.map createJavaStatement
typedStatementList))

let createJavaMain = function
          TypedStatementBlock(statementList) -> JavaMain(JavaBlock(List.map
createJavaStatement statementList))

let createJavaMainClass typedFunctionBlockList = function
        | TypedMainBlock(typedStatementList) -> JavaClass("Main", "", (createJavaMain
typedStatementList)::(List.map createJavaFunction typedFunctionBlockList))

let listPairToMap gropuName keyList valueList =
                JavaMap(
                        gropuName,
                        List.map string_of_int keyList,
                        List.map string_of_int valueList)

let listTablePairToMap gropuName keyList valueTable =
        JavaMap(
                gropuName,
                List.concat (List.map (fun element1 -> List.map (fun element2 -> string_of_int
element1 ^ "," ^ string_of_int element2) keyList) keyList),
                List.map string_of_int (List.concat valueTable))

let createJavaExtendedGroupClass = function
        | TypedGroupDeclaration(groupName, groupElements, additionTable, additiveInverseList)
->
                 (JavaClass(
                        groupName,
                        "Groups",
                        [JavaDefaultConstructor(
```

```
                                groupName,
                                JavaBlock(
                                        [JavaStatement(JavaConstant(listTablePairToMap
"additionTable" groupElements additionTable));
                                         JavaStatement(JavaConstant(listPairToMap
"additiveInverseList" groupElements additiveInverseList))]))]))
        | TypedRingDeclaration(groupName, groupElements, additionTable, additiveInverseList,
multiplicationTable) ->
                (JavaClass(
                        groupName,
                        "Rings",
                        [JavaDefaultConstructor(
                                groupName,
                                JavaBlock(
                                        [JavaStatement(JavaConstant(listTablePairToMap
"additionTable" groupElements additionTable));
                                         JavaStatement(JavaConstant(listPairToMap
"additiveInverseList" groupElements additiveInverseList));
                                         JavaStatement(JavaConstant(listTablePairToMap
"multiplicationTable" groupElements multiplicationTable))]))]))
        | TypedFieldDeclaration(groupName, groupElements, additionTable, additiveInverseList,
multiplicationTable, multiplicitiveInverseList, additiveIdentity) ->
                (JavaClass(
                        groupName,
                        "Fields",
                        [JavaDefaultConstructor(
                                groupName,
                                JavaBlock(
                                        [JavaStatement(JavaAssignment("additiveIdentity",
JavaConstant(JavaValue(JavaInt(additiveIdentity)))));
                                         JavaStatement(JavaConstant(listTablePairToMap
"additionTable" groupElements additionTable));
                                         JavaStatement(JavaConstant(listPairToMap
"additiveInverseList" groupElements additiveInverseList));
                                         JavaStatement(JavaConstant(listTablePairToMap
"multiplicationTable" groupElements multiplicationTable));
                                         JavaStatement(JavaConstant(listPairToMap
"multiplicitiveInverseList" (List.filter (fun element -> element <> additiveIdentity) groupElements)
multiplicitiveInverseList))]))]))

let createJavaAst = function
```

```
        TypedProgram(typedMainBlock, typedFunctionBlockList,
typedExtendedGroupBlockList) -> JavaClassList((createJavaMainClass typedFunctionBlockList
typedMainBlock)::(List.map createJavaExtendedGroupClass typedExtendedGroupBlockList))

(* Generate code from Java Ast *)
let generateJavaType logToJavaFile = function
        | JavaBooleanType -> logToJavaFile "boolean "
        | JavaIntType -> logToJavaFile "int "
        | JavaSetLiteralType -> logToJavaFile "PlatoSet<Object> "
        | JavaVectorLiteralType -> logToJavaFile "PlatoVector<Object> "
        | JavaCasesLiteralType -> raise(PlatoError("JavaCasesLiteralType has no direct java
object type"))
        | JavaNeutralType -> logToJavaFile "Object "

let generateJavaPrimitive logToJavaFile = function
        | JavaBoolean(booleanValue) -> logToJavaFile (string_of_bool booleanValue)
        | JavaInt(intValue) -> logToJavaFile (string_of_int intValue)

let rec generatePuts logToJavaFile mapName keyList valueList =
        (if List.length keyList > 0
        then (logToJavaFile (mapName ^ ".put(\"" ^ List.hd keyList ^ "\", \"" ^ List.hd valueList ^
"\");\n");
            generatePuts logToJavaFile mapName (List.tl keyList) (List.tl valueList))
        else ())

let generateJavaValue logToJavaFile = function
        | JavaValue(javaPrimitive) -> generateJavaPrimitive logToJavaFile javaPrimitive
        | JavaMap(mapName, keyList, valueList) ->
                logToJavaFile (mapName ^  " = new HashMap<String, String>();\n");
                generatePuts logToJavaFile mapName keyList valueList;
                logToJavaFile (mapName ^ " = Collections.unmodifiableMap(" ^ mapName ^ ")")

let rec generateJavaExpression logToJavaFile = function
        | JavaConstant(javaValue) -> generateJavaValue logToJavaFile javaValue
        | JavaVariable(stringValue) -> logToJavaFile stringValue
        | JavaReturn(expressionToReturn) ->
                logToJavaFile "return ";
                generateJavaExpression logToJavaFile expressionToReturn
        | JavaIf(javaExpression, javaBlock, javaElseIfList, javaElse) ->
                logToJavaFile "if(";
                generateJavaExpression logToJavaFile javaExpression;
                logToJavaFile ")";
                generateJavaBlock logToJavaFile javaBlock;
```

```
            ignore (List.map (generateJavaElseIf logToJavaFile) javaElseIfList);
            generateJavaElse logToJavaFile javaElse
        | JavaIfNoElse(javaExpression, javaBlock, javaElseIfList) ->
            logToJavaFile "if(";
            generateJavaExpression logToJavaFile javaExpression;
            logToJavaFile ")";
            generateJavaBlock logToJavaFile javaBlock;
            ignore (List.map (generateJavaElseIf logToJavaFile) javaElseIfList)
        | JavaAssignment(variableName, variableValue) ->
            logToJavaFile (variableName ^ "=");
            generateJavaExpression logToJavaFile variableValue
        | JavaVectorAssignment(variableName, indexValue, variableValue) ->
            logToJavaFile (variableName ^ ".set((");
            generateJavaExpression logToJavaFile indexValue;
            logToJavaFile (")-1, ");
            generateJavaExpression logToJavaFile variableValue;
            logToJavaFile (")");
        | JavaDeclaration(variableType, variableName, variableValue) ->
          generateJavaType logToJavaFile variableType;
            logToJavaFile (variableName ^ "=");
            (match variableValue with
              | Some(javaExpressionValue) -> generateJavaExpression logToJavaFile
javaExpressionValue
              | None -> () (* do nothing *))
        | JavaTernaryChain(casesExpressionsList, defaultCase) ->
            logToJavaFile ("(");
            ignore (List.map (fun (x, y) -> logToJavaFile ("(";
                                                generateJavaExpression
logToJavaFile y;
                                                logToJavaFile (")?(";
                                                generateJavaExpression
logToJavaFile x;
                                                logToJavaFile ("):"))
casesExpressionsList);
            logToJavaFile ("(";
            generateJavaExpression logToJavaFile defaultCase;
            logToJavaFile ("))") (* e^(PI*i)-1=0 *)
        | JavaCall(className, methodName, javaExpressionList) ->
            if (methodName = "")
            then (generateJavaParameters logToJavaFile javaExpressionList)
            else if (className = "")
                then (logToJavaFile (methodName ^ "(";
```

```
                                        generateJavaParameters logToJavaFile
javaExpressionList;
                                                logToJavaFile ")")
            else let invokationString = (if String.contains className '.' then className else
"(new " ^ className ^ "())")
                                        in logToJavaFile (invokationString ^ "." ^ methodName ^
"(");
                                    generateJavaParameters logToJavaFile javaExpressionList;
                                    logToJavaFile ")"
and generateJavaParameters logToJavaFile = function
        | [] -> ()
        | [first] -> ignore (generateJavaExpression logToJavaFile first)
        | first::rest -> ignore (generateJavaExpression logToJavaFile first);
                ignore (List.map (fun elem -> logToJavaFile ","; generateJavaExpression
logToJavaFile elem) rest)
and generateJavaElseIf logToJavaFile = function
        JavaElseIf(javaExpression, javaBlock) ->
                logToJavaFile "else if(";
                generateJavaExpression logToJavaFile javaExpression;
                logToJavaFile ")";
                generateJavaBlock logToJavaFile javaBlock
and generateJavaElse logToJavaFile = function
        JavaElse(javaBlock) ->
                logToJavaFile "else";
                generateJavaBlock logToJavaFile javaBlock
and generateJavaBlock logToJavaFile = function
        JavaBlock(javaStatementList) ->
                logToJavaFile "{\n";
                ignore (List.map (generateJavaStatement logToJavaFile)
javaStatementList);
                logToJavaFile "}\n"
and generateJavaStatement logToJavaFile = function
        JavaStatement(javaExpression) ->
                generateJavaExpression logToJavaFile javaExpression;
                (match javaExpression with
                        JavaIf(_, _, _, _) -> ()
                        | JavaIfNoElse(_, _, _) -> ()
                        | _ -> logToJavaFile ";\n")


let generateJavaFunctionParameter logToJavaFile = function
        Parameter(paramType, paramName) ->
                generateJavaType logToJavaFile (createJavaType paramType);
```

```
                        logToJavaFile paramName

let generateJavaMethod logToJavaFile = function
        | JavaMain(javaBlock) ->
                        (logToJavaFile "public static void main(String[] args) ";
                        generateJavaBlock logToJavaFile javaBlock)
        | JavaFunction(functionHeader, javaBlock) ->
                        (logToJavaFile "public static ";
                        (match functionHeader.returnType with
                                | VoidType -> ignore (logToJavaFile "void ")
                                | OtherType(returnType) -> ignore (generateJavaType
logToJavaFile (createJavaType returnType));
                                );
                        logToJavaFile functionHeader.functionName;
                        logToJavaFile "(";
                        (match functionHeader.parameters with
                                [] -> ()
                                | [first] -> ignore (generateJavaFunctionParameter logToJavaFile
first)
                                | first::rest ->
                                                ignore (generateJavaFunctionParameter
logToJavaFile first);
                                  ignore (List.map (fun elem -> logToJavaFile ",";
generateJavaFunctionParameter logToJavaFile elem) rest));
                        logToJavaFile ") ";
                        generateJavaBlock logToJavaFile javaBlock)
                | JavaDefaultConstructor(className, javaBlock) ->
                        (logToJavaFile ("public " ^ className ^ "()");
                         generateJavaBlock logToJavaFile javaBlock)

let removeFile fileName =
        if Sys.file_exists fileName
        then Sys.remove fileName

let generateJavaClass fileName = function
         JavaClass(javaClassName, javaSuperClassName, javaMethodList) ->
                        let fullClassName = (if javaClassName = "Main" then (javaClassName ^
"_" ^ fileName) else javaClassName)
                        in let fullFileName = fullClassName ^ ".java"
                                in removeFile fullFileName;
                           let logToJavaFile = logToFileAppend false fullFileName
                                  in let extendsString = (if javaSuperClassName = "" then "" else
("extends " ^ javaSuperClassName))
```

```
                                        in logToJavaFile "import java.util.*;\n\n";
                                            logToJavaFile (String.concat " " ["public class";
fullClassName; extendsString; "{\n"]);
                                        ignore (List.map (generateJavaMethod logToJavaFile)
javaMethodList);
                                logToJavaFile "}\n"

let generatePlatoBooleanClass =
        let logToBooleanClassFile = logToFileOverwrite false "Booleans.java"
        in logToBooleanClassFile booleanClassString

let generatePlatoIntegerClass =
        let logToIntegerClassFile = logToFileOverwrite false "Integers.java"
        in logToIntegerClassFile integerClassString

let generatePlatoSetLiteralsClass =
        let logToIntegerClassFile = logToFileOverwrite false "SetLiterals.java"
        in logToIntegerClassFile setLiteralsClassString

let generatePlatoSetClass =
        let logToIntegerClassFile = logToFileOverwrite false "PlatoSet.java"
        in logToIntegerClassFile platoSetClassString

let generatePlatoVectorLiteralsClass =
        let logToIntegerClassFile = logToFileOverwrite false "VectorLiterals.java"
        in logToIntegerClassFile vectorLiteralsClassString

let generatePlatoVectorClass =
        let logToIntegerClassFile = logToFileOverwrite false "PlatoVector.java"
        in logToIntegerClassFile platoVectorClassString

let generatePlatoGroupClass =
        let logToGroupClassFile = logToFileOverwrite false "Groups.java"
        in logToGroupClassFile groupClassString

let generatePlatoRingClass =
        let logToRingClassFile = logToFileOverwrite false "Rings.java"
        in logToRingClassFile ringClassString

let generatePlatoFieldClass =
        let logToFieldClassFile = logToFileOverwrite false "Fields.java"
        in logToFieldClassFile fieldClassString
```

```
let generatePlatoClasses =
        generatePlatoBooleanClass;
        generatePlatoIntegerClass;
        generatePlatoSetLiteralsClass;
        generatePlatoSetClass;
        generatePlatoVectorLiteralsClass;
        generatePlatoVectorClass;
        generatePlatoGroupClass;
        generatePlatoRingClass;
        generatePlatoFieldClass

let generateJavaCode fileName = function
          JavaClassList(javaClassList) ->
                    generatePlatoClasses;
                    ignore (List.map (generateJavaClass fileName) javaClassList)

let compile fileName =
  let lexbuf = Lexing.from_channel (open_in fileName)
        in let programAst = Parser.program Scanner.token lexbuf
          in logProgramAst programAst;
                  let programSast = checkProgram emptyGlobalEnvironment programAst
                  in logProgramSast programSast;
                              let javaClassListAst = createJavaAst programSast
                    in logJavaClassListAst javaClassListAst;
                                        let basename =  Filename.basename fileName
                                        in if (String.sub basename ((String.length basename) -
4) 4) = ".plt"
                                                              then generateJavaCode
(Filename.chop_extension basename) javaClassListAst
              else raise(PlatoError("Invalid file extension"))
let _ = compile Sys.argv.(1)
```